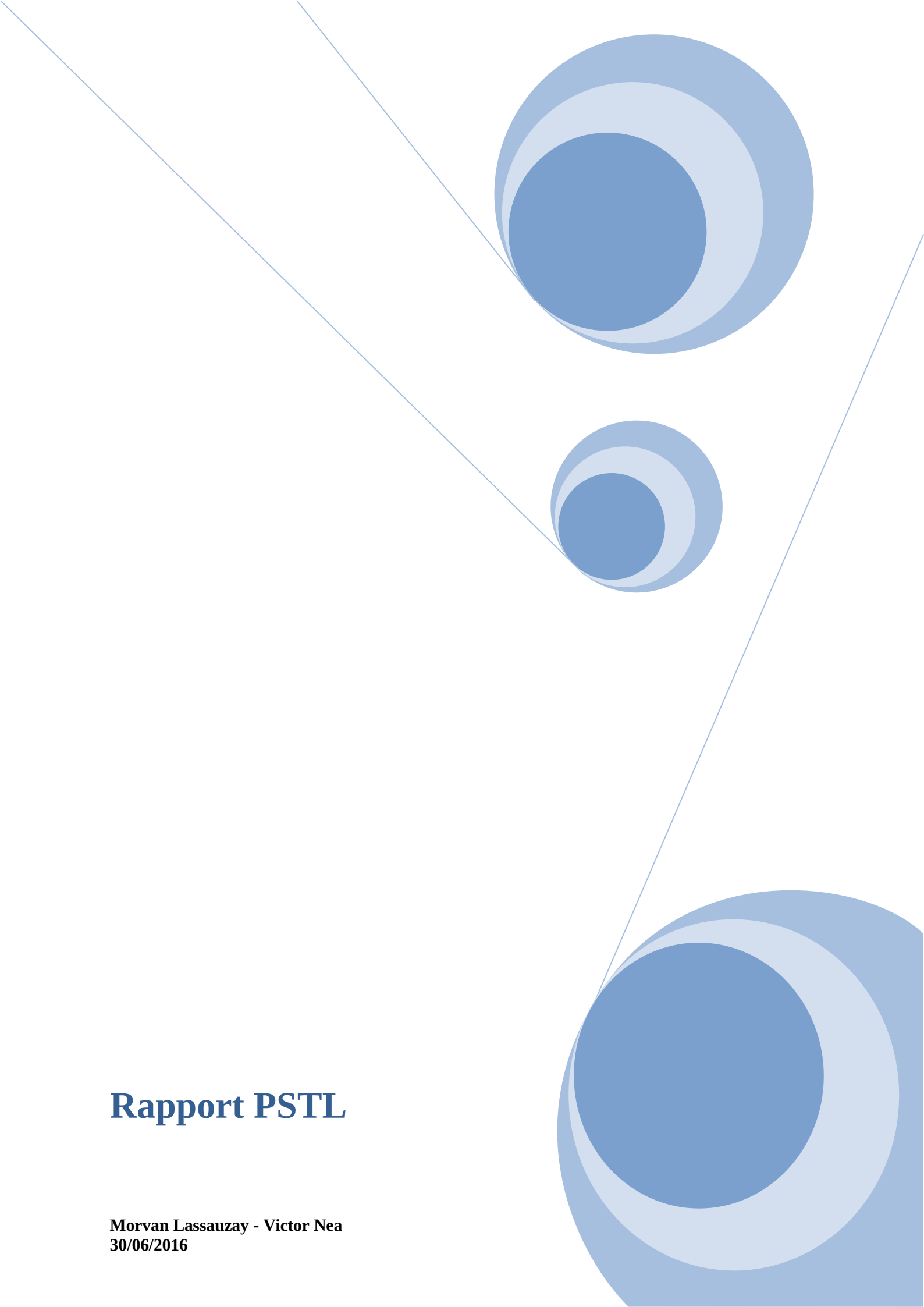

Rapport PSTL

Morvan Lassauzay - Victor Nea
30/06/2016

Sommaire

Introduction

Dans le cadre de notre formation en 1ère année de Master spécialité STL, nous avons été amenés à développer une application nommée dotEd pour l'UE PSTL. Cette application a pour objectif premier de permettre graphiquement la manipulation et l'édition de petits graphes d'une centaine de nœuds au maximum, afin de simplifier l'utilisation de ces graphes. En plus de cette gestion graphique l'application doit offrir en parallèle la possibilité d'éditer des graphes via le format texte de description de graphe DOT. Il est donc question de fournir à l'utilisateur une interface en deux parties, l'une graphique, l'autre textuelle, où chaque interaction sur l'une de ces deux parties sera retransmise instantanément sur l'autre. Naturellement l'application se doit également de proposer la possibilité d'importer et d'exporter des fichiers au format DOT, si possible sans altération des éléments du fichier tant par leur contenu que par leur emplacement.

Le format DOT met à disposition de ses utilisateurs tout une gamme d'attributs influant sur le rendu graphique des graphes. Bien évidemment il est impossible de traiter l'ensemble de ces attributs dans le temps qui nous est imparti. Cependant l'application doit être en mesure d'accepter à l'import des fichiers disposant d'éléments encore non traités visuellement. Afin d'assurer la maintenabilité au travers de l'ajout de nouveaux éléments, ou éventuellement d'un réaménagement plus important comme un changement d'interface graphique, l'application se doit de plus d'être modulaire tout en gardant une organisation claire et efficace.

I – Etude préalable

1. Le format DOT

Le langage DOT est un langage de description de graphe dans un format texte (extension dot). Il est donc possible de définir des nœuds, des arrêtes, des sous-graphes et de leur définir des attributs tel qu'un label, une couleur ou encore une forme. On peut dès lors voir apparaître deux phases dans le développement de notre application, la première consistant à offrir la possibilité d'éditer de simples graphes composés de nœuds et d'arêtes nus, et la seconde consistant à ajouter les différents éléments du format que sont les sous-graphes et les divers attributs. Pour bien visualiser la syntaxe du format Voici un petit exemple de graphe DOT utilisant les éléments décrit précédemment :

```
graph mon_graphe {
        A [label = noeudA];
        B;
        subgraph subG {
                C [color=yellow, shape=square];
                D;
        }
        A--B;
        A--C [label=transition];
        D--B;
}
```

Même si cela n'est pas obligatoire il nous a été dit de considérer que chaque instruction se terminait par un point-virgule (à l'exception des sous-graphes qui utilisent les accolades). On voit alors apparaître une structure globale relativement simple autour de laquelle il sera certainement possible de généraliser. Pour plus d'information quant à la syntaxe du format DOT vous trouverez joint en annexe la grammaire du langage.

2. Veille concurrentielle

Graphviz :

Graphviz (Graph Visualization Software) est le logiciel incontournable lorsque l'on parle du format DOT, en effet on peut le qualifier de logiciel officiel du langage. A l'aide de GraphViz, il est possible d'importer et d'éditer textuellement des fichiers DOT afin de visualiser le graphe correspondant, puis de sauvegarder ce rendu graphique sous forme d'image.

Voici un exemple d'un fichier DOT et de sa représentation graphique avec GraphViz :

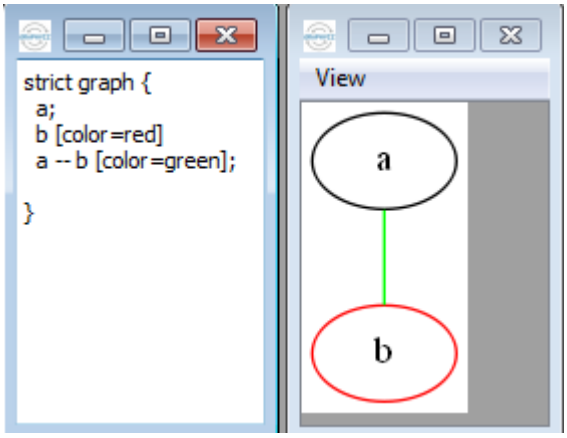

Figure 1 : Représentation graphique d'un fichier DOT

GraphViz est un outil très puissant pour la génération graphique de graphe. Cependant il n'est pas possible de faire des actions (création, édition ou suppression d'un élément) graphiquement. Le seul moyen est donc de modifier le fichier DOT puis régénérer le graphe. Notre logiciel offre donc un réel avantage pour construire et visualiser rapidement un graph simple.

A noter qu'à l'adresse http://graphviz.herokuapp.com/ il est possible d'utiliser Graphviz directement via un navigateur web et de disposer d'une interface semblable à celle que l'on souhaite pour notre application. Cependant l'édition graphique n'est toujours pas au menu.

DotEditor :

DotEditor offre de nombreuses possibilités intéressantes. Il permet d'éditer des graphes via une interface en cliquant sur des boutons et en remplissant des formulaires. Moins intuitif et rapide à utiliser que notre application il permet néanmoins d'établir des graphes très détaillés sans devoir tout rédiger textuellement. De plus la possibilité de visualiser le texte correspondant au graphe est également présente et il est même possible de modifier directement ce texte puis de visualiser le graphe résultant. Malgré tout il est impossible de choisir soi-même la position des nœuds contrairement à notre application. DotEditor est un bon outil duquel il est possible de s'inspirer tout en gardant bien à l'esprit qu'il ne répond pas totalement aux attentes de l'application que l'on souhaite développer.

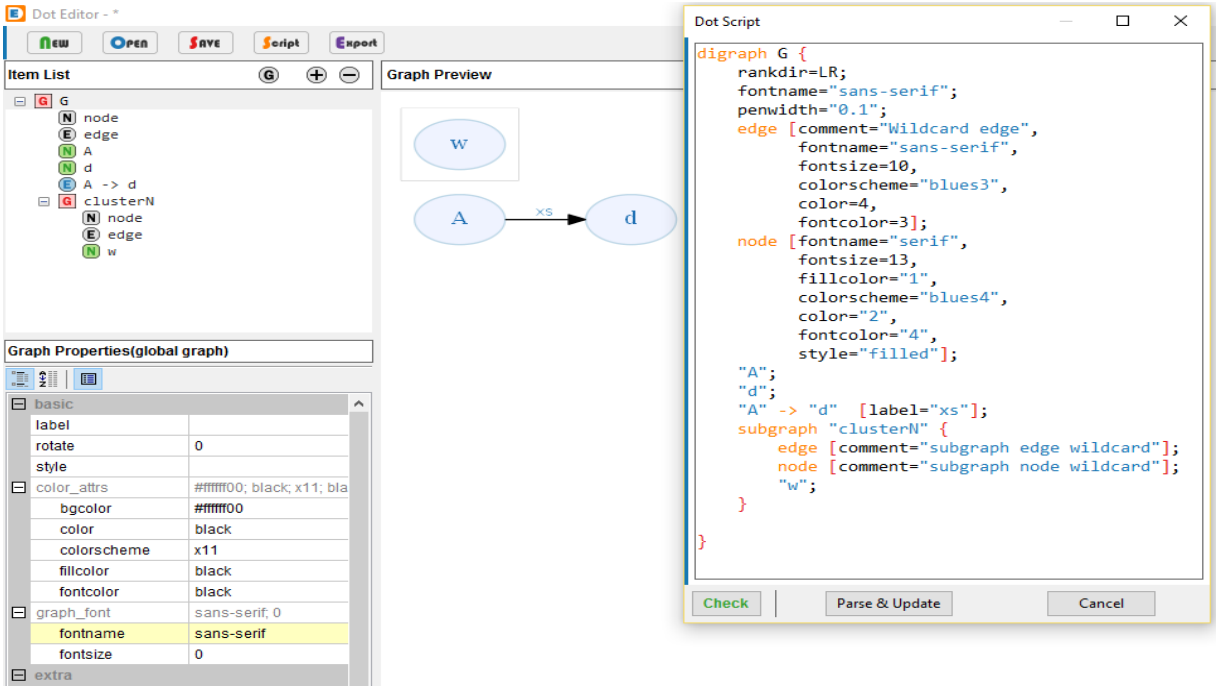

Figure 2 : Capture d'écran du logiciel DotEditor

Autres :

Nous avons trouvé de nombreux autres logiciels se disant être des éditeurs de graphes au format DOT, cependant la totalité ont été abandonnés en court de route ou sont tous simplement incomplets ou encore bogué à l'installation. On peut retrouver dans cette liste les logiciels dotty, KGraphEditor, grappa, graphedit.

3. Vue d'ensemble de l'application espérée

Nous avons désormais toutes les informations nécessaires pour disposer d'une bonne vue d'ensemble de l'application que nous souhaitons développer. Voici donc la très simple maquette sur laquelle nous nous sommes basés :

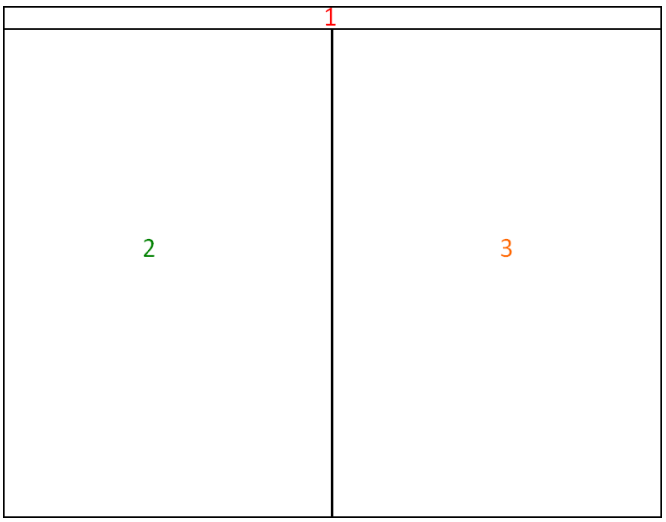

Figure 3 : Maquette de l'application

1: Barre de menus avec des boutons pour l'import, l'export et la création d'un nouveau graphe.
2: Vue graphique pour afficher le graphe
3: Vue textuelle pour afficher la description textuelle du graphe au format DOT
Ceci avec la possibilité de redimensionner les deux vues comme bon nous semble. De plus il serait appréciable de disposer d'un système permettant de surligner dans le texte un élément sélectionné sur la vue graphique. Bien entendu le tout toujours accompagné de la répercussion instantanée sur chaque vue des modifications effectuées sur l'autre vue.

4. Outils choisis

Tout d'abord, nous avons choisis le langage Python pour développer notre application. Nous avons choisi ce langage pour plusieurs raisons :

- la découverte d'un nouveau langage
- la portabilité : Puisque Python est multiplateforme, il sera facile de distribuer l'application.
- la gestion de la mémoire : Python dispose d'un système de gestion automatique de la mémoire, il n'est donc pas nécessaire de se soucier de la libération des ressources ce qui permet d'éviter les fuites de mémoire et d'avoir une application robuste.
- la documentation : Python est très bien documenté, il est donc facile de trouver ce que l'on cherche.
- la bibliothèque standard : Elle est très vaste ce qui permet de ne pas réinventer la roue et d'accéder de manière très simple à des fonctionnalités élaborées.
- PEP (Python Enhancement Proposal) : Nous en reparlerons en fin de rapport.
- C'est un langage orienté objet ce qui nous permettra de gérer plus simplement et clairement nos différents éléments graphiques.

En ce qui concerne la bibliothèque graphique, nous avons décidé d'utiliser PyQt (qui est la version Python de Qt) qui est également une bibliothèque que nous ne connaissions pas. PyQt a été développée avant tout dans le but de créer des interfaces utilisateurs ce qui correspond parfaitement à notre cas. De plus elle est très bien documentée et est multiplateforme. C'est également une des seules si ce n'est la seule bibliothèque graphique à disposer d'une scène pour manipuler des objets graphiques. Il n'est donc pas nécessaire de recoder toute la partie qui permet de gérer l'ajout, le déplacement ou la surpression d'un élément graphique de la scène.

Il nous a également fallu chercher un outil pour nous permettre de parser un texte représentant un graphe et en extraire son contenu. Grâce à nos enseignants nous avons trouvé très facilement l'outil Pydot, un très bon parseur de texte au format Dot et simple d'utilisation.

Afin de partager facilement le code source nous avons utilisé Git, ceci nous permettant d'avoir une sauvegarde en cas de problème et d'avoir un contrôle de version.

Enfin nous avons utilisé l'IDE LiClipse qui est une version customisée d'Eclipse. Il n'est pas primordial d'utiliser un IDE pour développer en python mais LiClipse apporte plusieurs avantages :

- Le plugin PyDev qui permet de développer en Python.
- Le plugin EGit qui permet d'utiliser Git directement dans LiClipse via une interface graphique.
- Un débogueur bien utile en cas de problème.
- L'auto-complétion.

II – Une application graphique

Cette partie a pour but de décrire l'ensemble du fonctionnement et du contenu de l'application. Pour cela nous présenterons les différents éléments dans l'ordre dans lequel ils ont été abordés lors du développement.

1. Structure et fonctionnement général

Le but de notre projet étant de développer une application avec une interface graphique, nous avons dû réfléchir à la façon dont nous allions organiser notre projet pour qu'il soit facilement maintenable, efficace et évolutif.

1.1 Architecture MVC

Développer une application graphique était en quelque sorte une nouveauté pour nous. Nous avons donc dû étudier et implémenter une architecture MVC (Model-View-Controller) nous permettant de séparer le code source en 3 différents composants :

- Modèle : Le modèle représente l'ensemble des données sur lesquelles on veut travailler, soit dans notre cas, l'ensemble des éléments du graphe. Il doit offrir des méthodes, si possible précise, pour pouvoir mettre à jour les données (ajout/édition/suppression d'un nœud par exemple) ou bien tout simplement les récupérer.
- Vue : La vue est le résultat du modèle, c'est ce que voit l'utilisateur à l'écran. C'est dont elle qui nous permet de représenter notre graphe sous forme textuelle ou graphique. Mis à part afficher le résultat du modèle, la vue doit également être en mesure de capter les interactions entre le client et l'interface (comme les clics de souris, la saisie d'un texte, etc…).

- Contrôleur : Le contrôleur assure la communication entre la vue et le modèle lors des mises à jour et permet si besoin d'effectuer différents calculs sur ces mises à jour, bien qu'en ce qui nous concerne cela ne soit presque jamais le cas.

L'avantage de cette architecture est l'indépendance de chaque composant. Cette séparation du code source permet d'avoir un projet facilement maintenable. Une approche naïve aurait été de développer l'application en fusionnant le modèle et la vue. Cependant cette solution pose un problème majeur : Imaginons qu'un jour nous décidions de changer de bibliothèque graphique, les deux composants auraient alors été liés et il aurait été très difficile de faire toutes les modifications nécessaires dans le code.

Pour la mise en place d'une architecture MVC, il existe plusieurs façons de définir et mettre en relation les différents composants. Nous souhaitions pouvoir ajouter, retirer ou modifier une vue en intervenant uniquement sur le code la concernant. Pour cela un couple vue/contrôleur pour chacune de nos vues a été créé. De cette manière il a été possible de travailler en isolation d'une part sur la vue textuelle et d'autre part sur la vue graphique. De plus, afin de pouvoir relier au modèle cet ensemble de couples de taille potentiellement variable, le design pattern Observer a été appliqué. Le diagramme de classes fourni en annexe permet de disposer d'une représentation explicite de l'architecture obtenue.
Ainsi, comme il est possible de le voir sur le diagramme, chaque contrôleur vient s'inscrire en observateur du modèle. De cette manière chacun d'eux est directement averti par le modèle lorsque ce dernier subit une modification. Pour faire cela le modèle dispose d'une méthode notify() envoyant les informations du nœud ou de l'arête ayant été créé, supprimé ou bien modifié. Nous reviendrons par la suite sur les mécanismes de mise à jour du modèle suite à une demande de l'utilisateur.

1.2 Les différents packages

Afin de regrouper les éléments possédant une logique semblable le projet a été découpé en différents packages :

- app : Contient la classe principale de l'application permettant d'instancier les différents composants. Malgré les nombreuses références entre composants les mécanismes d'instanciation ont été pensés de manière à garder très simple le code de cette classe.
- controller : Englobe l'ensemble des différents contrôleurs. On y trouve la classe « Controller » servant d'abstraction au contrôleur respectif de nos vues graphique et textuelle. A noter que le contrôleur de la fenêtre principale n'hérite pas de « Controlleur » ce premier n'étant pas concerné par les mises à jour du modèle.
- enumeration : Ce package contient l'ensemble des différentes énumérations utiles à la gestion des nœuds et arêtes, cela allant du nom des attributs pris en compte aux différents modes de

mise à jour. L'utilisation de ces énumérations nous permet de centraliser et d'uniformiser les différents termes utilisés dans l'ensemble du programme.

- model : Comprend simplement les ressources qui concerne le Modèle.
- observer : Contient les classes « Observer » et « Subject » utile au design pattern Observer.
- utils : Contient une classe de méthodes utiles aux nœuds et une autre pour les arêtes. On y trouve par exemple une méthode pour définir l'identifiant d'une arête ou bien une méthode pour vérifier la validité des attributs d'un nœud.
- view : Ce package comprend l'ensemble des différentes vues et des différents éléments graphiques pour la représentation des nœuds et des arêtes. Le packages « widget » correspond aux vues (on remarquera les mêmes similitudes hiérarchiques que pour le package « controller »).

1.3 Modèle et mises à jour

Pour bien comprendre le fonctionnement général de l'application il faut dans un premier temps s'intéresser à la modélisation du modèle. Notre modèle est composé d'un dictionnaire de nœuds et d'un dictionnaire d'arêtes. Chaque nœud et arête possède un identifiant qui lui est propre et est donc unique. On utilise cet identifiant comme clé dans les dictionnaires pour accéder aux informations concernant le nœud ou l'arête qui lui correspond. A l'heure actuel, outre un identifiant, un nœud est composé d'un dictionnaire d'attributs. En plus de permettre de stocker le label et la position du nœud qui sont les deux attributs gérés actuellement par l'application, ce dictionnaire nous permet de conserver tous les attributs même si ceux-ci ne sont pas encore gérés graphiquement par l'application. Le traitement futur de ces attributs s'en voit ainsi nettement simplifié. Les arêtes quant à elle ne sont constituées que d'un nœud source et d'un nœud destination. Le format Dot ne les exclut pas de posséder des attributs mais ceux-ci se voient être bien moins utiles que ceux dédiés aux nœuds (à l'exception du label selon le besoin). Cette structure est très importante car elle va déterminer le contenu de chaque composant de l'application.

Les mises à jour aussi bien du modèle que des vues sont reparties en trois types : ajout, suppression, ou édition (modifications des attributs), qui peuvent s'appliquer aussi bien aux nœuds qu'aux arêtes, à l'exception de l'édition les arêtes ne disposant pas encore d'attribut. Ainsi la vue graphique, la vue textuelle, et le modèle, disposent chacun de cinq méthodes pour réaliser ces cinq actions différentes. Pour effectuer les mises à jour du modèle vers les vues ou d'une vue vers le modèle il nous suffit d'envoyer l'identifiants de l'élément concerné ainsi qu'un dictionnaire contenant les informations utiles. Une clé unique pour chaque type d'information étant stockée dans nos différentes énumérations, il nous est simple de les récupérer tout au long de la chaine sans souci de cohérence. La consultation des contrôleurs, dont le code est très court, permet de voir explicitement la mise en place de cette gestion des mises à jour.

2. Interfaces graphiques

Dans cette partie nous allons nous intéresser à la gestion des différentes vues de l'application. Pour rappel, nous avons utilisé la bibliothèque graphique PyQt. Ainsi lorsque nous parlerons d'une classe QClassName, cela signifiera que l'on parle d'une classe de PyQt.

2.1 La vue principale : MainWindow

Voici un aperçu du logiciel :

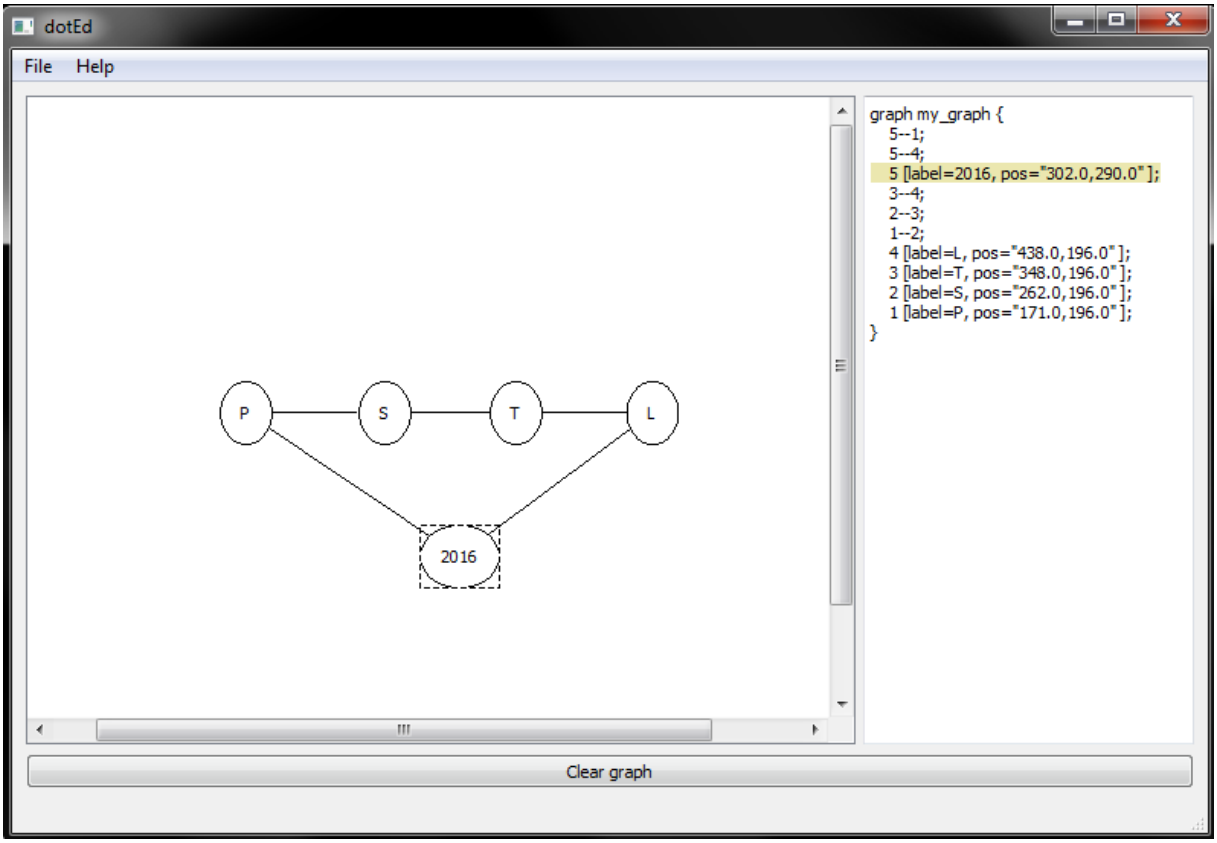

Figure 4 : Aperçu de l'application

La vue du logiciel est générée grâce à la classe MainWindow. Comme on peut le voir sur l'image ci-dessus, elle contient nos deux vues principales GraphicsGraphView et TextGraphView. Ces deux vues sont séparées par une barre de séparation (classe QSplitter) qui permet d'agrandir ou de rétrécir une vue ou l'autre. Elle contient également une barre de menus qui permet d'effectuer différentes actions comme importer ou exporter un fichier DOT. Le bouton "Clear Graph" permet simplement d'effacer le graphe dans les deux vues.

La vue MainWindow est modulaire, c'est à dire qu'il est facile d'y ajouter de nouvelles vues si nous le souhaitons. Elle dispose en effet d'une méthode addWidget(...) qui permet d'ajouter une vue. Toutes les vues que l'on voudra ajouter devront hériter de notre classe View et d'une classe Qt qui hérite de la classe QWidget.

C’est à partir de cette vue que nous allons pouvoir demander l’import et l’export de fichier. Ceci est réalisé à l’aide d’une boite de dialogue fourni toute prête par Qt permettant d’accéder au système de fichiers. Nous avons configuré un filtre pour ne visualiser que les fichiers dot si souhaité et accélérer la recherche. Pour l’export nous récupérons simplement le texte contenu dans la vue textuelle et le copions dans le fichier choisi. Pour l’import nous utilisons l’outils Pydot pour parser le fichier texte et récupérer l’ensemble des nœuds et arêtes contenus dans celui-ci.

2.2 La vue graphique : GraphicsGraphView

C’est cette vue qui va s'occuper d'afficher le graphe. Elle permet donc d'ajouter, éditer et supprimer graphiquement des nœuds et des arêtes.

2.2.1 Description détaillée

Voici un diagramme de classe de GraphicsGraphView:

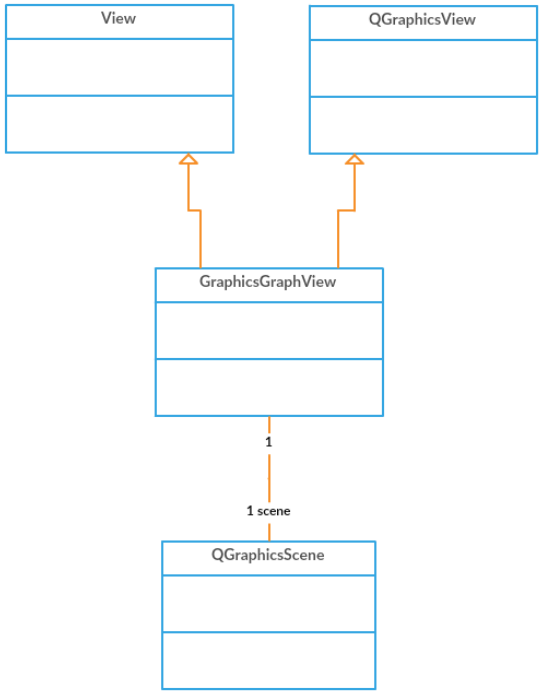

Figure 5 : Diagramme de classes de GraphicsGraphView

GraphicsGraphView hérite de la classe View et de la classe QGraphicsView (qui hérite de la classe QWidget). Elle respecte donc les critères pour pouvoir être ajoutée à MainWindow avec la méthode addWidget(...). Comme on peut le voir sur le diagramme ci-dessus, GraphicsGraphView dispose d'une QGraphicsScene. C'est cette scène qui s'occupe de l'affichage des éléments graphiques et qui nous permet de les manipuler. Cette scène dispose de plusieurs évènements, comme le clic souris, le déplacement de la souris, et bien d’autres. On peut donc redéfinir ces évènements afin d'effectuer les actions souhaitées.

2.2.2 Gestion des nœuds

Nous allons ici nous intéresser au lien des nœuds graphiques avec la classe GraphicsGraphView.

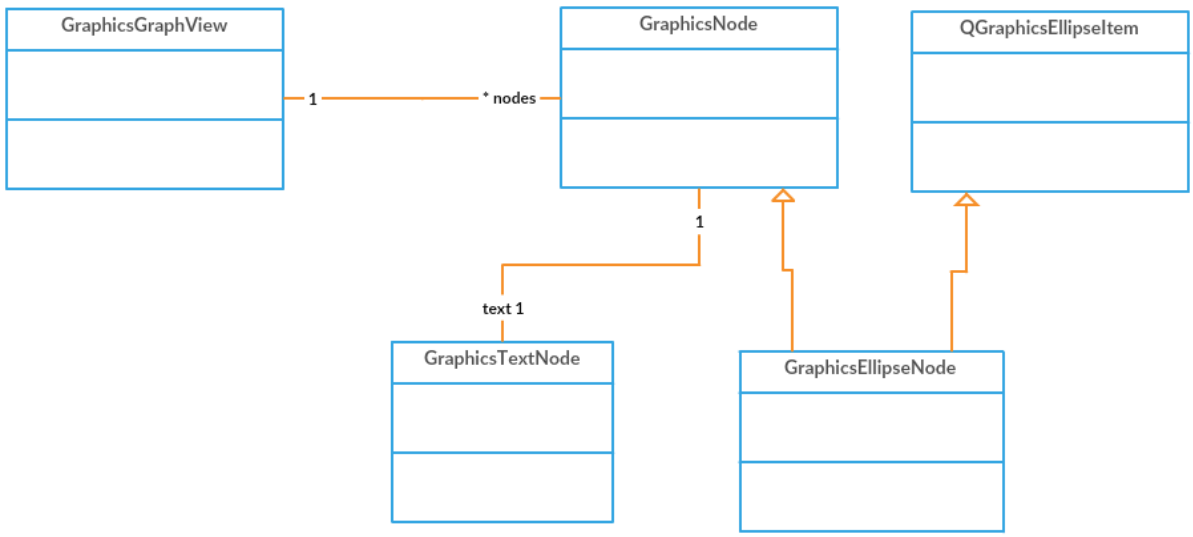

Figure 6 : Diagramme de classes de GraphicsGraphView avec nœuds

Pour ce qui est de la gestion des nœuds graphiques, nous avons créé une classe mère GraphicsNode qui regroupe les attributs et les méthodes communes à tous les nœuds graphiques que l'on pourra créer. Chaque GraphicsNode dispose d'une GraphicsTextNode qui permet simplement d'afficher le texte (attribut label de DOT) du nœud. Pour le moment, l'application ne gère que la forme ellipse d'où la classe GraphicsEllipseNode. Lorsque que l'on veut ajouter une nouvelle forme, il faut que la nouvelle classe hérite de la classe GraphicsNode ainsi que d'une classe PyQt de type QAbstractGraphicsShapeItem. En l'occurrence, il n'existe que les formes polygone, ellipse et rectangle qui ont déjà été implémentées par PyQt. Si l'on souhaite ajouter la forme étoile par exemple, il faudra la programmer soi-même.

2.2.3 Gestion des arêtes

Nous allons ici nous intéresser au lien des arêtes graphiques avec la classe GraphicsGraphView.

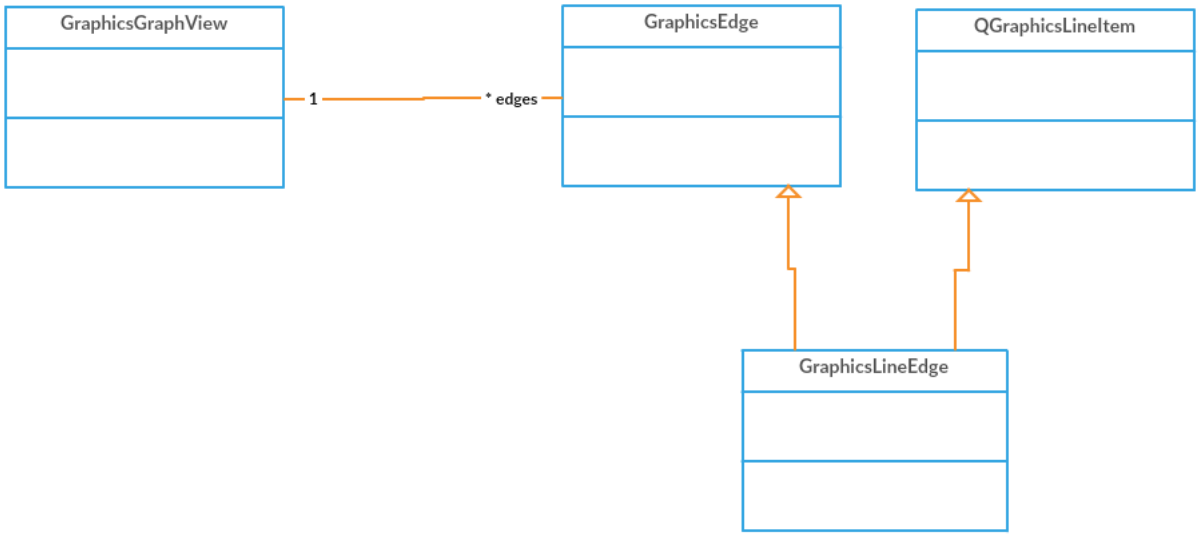

Figure 7 : Diagramme de classes de GraphicsGraphView avec arêtes

Le mécanisme de gestion des arêtes ressemble fortement à celui des nœuds. Tout comme pour les nœuds graphiques, les arêtes disposent d'une classe mère GraphicsEdge qui regroupe les attributs et méthodes communes à toutes les arêtes graphiques que l'on voudra créer. Seules les arêtes linéaires (classe QGraphicsLineItem) sont gérées pour le moment. Lorsque l'on souhaite ajouter une nouvelle forme d'arête il faut la faire hériter de GraphicsEdge et également d'une classe PyQt de type QGraphicsItem. Cependant nous avons constaté que seules les lignes ont déjà été implantées par PyQt. Si l'on veut avoir une arête courbée, il faudra programmer soit même cet élément graphique.

2.2.4 La gestion des évènements

Dans PyQt, il est possible de gérer de nombreux évènements comme le clic de souris, le double clic de souris, la pression d'une touche, etc. Il est donc très intéressant de pouvoir utiliser ces évènements pour exécuter certaines actions au lieu de passer par des menus et cliquer sur des boutons.

Tout d'abord, il a fallu que nous listions l'ensemble des actions que nous voulions gérer et regarder si chaque action avait un évènement PyQt correspondant :

- Double clic de souris sur la scène pour créer un nœud.
- CTRL + clic de souris (maintenu) sur un nœud puis mouvement de souris pour le déplacer.
- Double clic de souris sur un nœud pour éditer son label.
- Clic de souris (maintenu) sur un nœud puis relâchement sur un autre nœud pour créer une arête.
- Clic de souris sur un nœud ou une arête puis pression de la touche clavier "suppr" pour supprimer l'élément.

Voici la table de correspondance :

Actions	**Evènements PyQt**

Clic de souris (pression)	mousePressEvent
Clic de souris (relâchement)	mouseReleaseEvent
Double clic de souris	mouseDoubleClickEvent
Déplacement de souris	mouseMoveEvent
Pression d'une touche du clavier	keyPressEvent

Il suffit alors de surcharger ces méthodes (évènements) et d'écrire le code qui va effectuer l'action voulue.

Remarque :

Il faut noter que ces événements "ne sont pas définis globalement".

Si l'on veut gérer le clic de souris sur une arête, il faudra redéfinir la méthode mousePressEvent dans la classe GraphicsEdge.

Si l'on veut gérer le clic souris sur un nœud, il faudra redéfinir la méthode mousePressEvent dans la classe GraphicsNode.

2.3 La vue textuelle : TextGraphView

Cette vue ne dispose pas réellement d'élément graphique, elle se contente d'hériter de la classe QTextEdit permettant d'afficher un texte. Cependant elle constitue sans doute la partie la plus délicate de l'application. En effet nous avons longtemps chercher comment d'une part récupérer les modifications faites par l'utilisateur (cette vue s'utilisant comme un éditeur de texte) et d'autre part comment afficher correctement les modifications faites par l'utilisateur à partir de la vue graphique.

2.3.1 Modification via le texte

Pour ce cas, l'outil Pydot que nous vous avons présenté dans la première partie de ce rapport se montre idéal. Cependant parser le texte contenu par notre vue pour en extraire les nœuds et les arêtes formant le graphe n'est pas suffisant. Un mécanise nous permettant de reconnaître exactement quelles modifications ont été apportées est de plus nécessaire. Pour cela nous avons choisi de garder une sorte de copie du modèle dans notre vue. Ainsi à chaque fois que l'utilisateur effectue une modification sur le texte, il nous est possible de le parser à nouveaux et de comparer les nouveaux éléments obtenus avec les anciens éléments. Après quelques vérifications nous sommes donc en mesure d'en déduire quels nœuds et quelles arêtes ont été ajoutés, supprimés ou simplement modifiés.

Plusieurs questions se pose alors : Quand considérer qu'un utilisateur a fini d'effectuer une modification sur le texte ? Comment vérifier qu'une modification est juste et que faire si ce n'est pas le cas ? Pour la première nous avons choisi de considérer que la fin d'une modification correspondait à une perte de focus de la vue textuelle. Ainsi à chaque fois que l'utilisateur clic en dehors de cette vue alors qu'il avait le curseur sur cette vue nous effectuons les opérations décrites dans le paragraphe précédent. Cependant avant d'effectuer toute opération qui risquerait de nuire à l'état du programme nous devons nous assurer que le texte entré par l'utilisateur est valide. Pour cela nous utilisons une

nouvelle fois Pydot qui renvoie un statut particulier si une erreur est apparue au cours du parsage. Si c'est le cas nous faisons alors apparaître une fenêtre d'erreur puis nous replaçons le curseur sur le texte pour indiquer à l'utilisateur que nous attendons un texte valide avant de pouvoir continuer. A noter que Pydot ne relève pas toutes les erreurs dans la constitution d'un texte décrivant un graphe. Certaines erreurs sont simplement visuelles, comme un attribut portant un nom dérisoire par exemple, et ont donc été laissées, sous peine de devoir réécrire un parseur pour les traiter. D'autres cependant, comme une valeur fausse pour un attribut au nom valide, peuvent poser problème. C'est pourquoi nous avons dans le package « utils » défini une méthode permettant de vérifier la validité des attributs concernés. Si un des attributs est mal formé nous ouvrons alors également une fenêtre d'erreur pour le signaler à l'usager.

Remarque :

Lorsqu'un attribut est supprimé, sa valeur est remplacée à sa valeur par défaut. Ainsi par exemple, un nœud dont l'attribut de position se voit supprimé est replacer en position (0,0).

2.3.2 Insertion d'une modification dans le texte

Certainement le point le plus ardu de l'application. Dans un premier temps nous avions choisi pour répondre à ce problème de reconstruire le texte à chaque modification envoyée par le modèle. Pour cela nous mettions dans un premier temps à jour la copie du modèle contenue dans la vue puis nous réécrivions le texte à partir de cette copie. Cette technique ne permettait cependant pas de garder le texte dans l'état dans lequel l'utilisateur l'avait défini.

Nous avons donc dans un second temps développé une nouvelle version de cette vue permettant de conserver le texte exactement comme l'utilisateur souhaitait le définir. Pour cela nous récupérons à l'import (s'il y a import) le texte contenu dans le fichier importé sans le modifier puis nous le plaçons dans notre vue. Ensuite lors d'une modification sur le texte, la technique de la sous-partie précédente est appliquée. Dans le cas d'une modification survenu sur la vue graphique il nous faut alors être capable de réécrire exactement au bon endroit dans le texte la modification en question. Pour cela nous avons une nouvelle fois recours à Pydot pour effectuer le processus qui va suivre. Dans un premier temps nous découpons notre texte sur chaque point-virgule afin de récupérer chaque instruction de manière isolée. Ensuite nous parsons ces instructions une à une dans l'ordre du document jusqu'à rencontrer le nœud ou l'arête ayant subi une modification. Pour chaque instruction parcouru nous incrémentons un compteur de la taille de cette instruction en nombre de caractères. De cette manière, lorsque nous rencontrons l'instruction du nœud ou de l'arête qui nous intéresse, nous disposons exactement du caractère à laquelle cette instruction débute dans le texte. S'il s'agit d'une suppression il nous suffit de supprimer les n caractères suivant, n représentant la taille de l'instruction. S'il s'agit d'une modification nous recherchons dans l'instruction du nœud la place des attributs modifiés grâce aux expressions régulières. Une fois cela effectué nous recherchons dans la copie du modèle appartenant à la vue l'ancienne valeur de l'attribut correspondant. Nous récupérons ainsi la

taille n de cette valeur, puis supprimons le bon nombre de caractères dans le texte en fonction de n. Il ne nous reste ensuite plus qu'à recopier la nouvelle valeur de l'attribut à la bonne place.

Lors d'un ajout le nœud ou l'arête est simplement écrit en tête de texte : savoir à quel endroit l'utilisateur souhaite placer cet élément dans le texte étant impossible.

(Essayer d'écrire l'algorithme de la modification des attributs d'un noeud ?)

Vous avez certainement remarqué sur la figure 4 qu'une instruction était surlignée en jaune. Cette instruction correspond au nœud ou à l'arête qui est sélectionné sur la vue graphique. Pour faire cela nous faisons passer un message de la vue graphique à la vue textuelle grâce à différents appels de méthode (voir getFocus(), onSelectItem(), highlightItem()) et ceci à chaque fois qu'un élément est sélectionné dans la vue graphique. C'est cette fonctionnalité qui explique la référence que possède le contrôleur de la vue graphique vers celui de la vue textuelle. Une fois la demande reçue, la vue textuelle recherche dans le texte l'élément sélectionné grâce à son identifiant. Pour cela on utilise le même principe que lors de la modification d'un nœud. En d'autres termes on utilise Pydot pour retrouver l'instruction qui correspond à l'élément sélectionné puis on surligne le texte sur toute la longueur de cette instruction.

III - Maintenabilité et distribution

1. Exemple d'extension

2. PEP, licence, packaging

La PEP un document de conception qui décrit l'ensemble des conventions et bonnes pratiques pour coder en Python. En respectant ces normes, le code sera plus maintenable et une nouvelle personne qui souhaite travailler sur ce projet aura beaucoup plus de facilité à le comprendre.

Pour notre application nous avons choisi d'utiliser la licence GNU GPL (GNU General Public License) qui offre de nombreux avantages. Le premier de ces avantages est qu'il s'agit d'une licence bien connu du public et qu'il est très simple de se renseigner à son sujet, ce qui offre une certaine assurance de son respect sauf infraction volontaire. Un autre de ces avantages est qu'elle permet d'utiliser et de modifier le code source comme bon nous semble, ce qui est appréciable, surtout dans un logiciel pouvant subir de nombreux ajouts comme c'est le cas pour le nôtre (Gestion graphique des attributs et des éléments particuliers du langage DOT). Le point le plus intéressant de cette licence à notre sens est qu'elle oblige la redistribution à la communauté dès la moindre distribution d'une version modifiée, bien que cela soit dans les faits difficilement vérifiable. Le seul élément gênant pour cette licence dans notre cas, est qu'elle permet la revente des versions modifiées. Cependant une fois une version achetée par un utilisateur, celle-ci peut être redistribuée gratuitement. Cela rend donc difficile la génération de profit à l'insu des utilisateurs, sans l'exclure toutefois. Bien entendu nous sommes conscients que la plupart si ce n'est la totalité de ces cas ne se produirons pas avec notre logiciel mais il nous a semblé intéressant d'étudier le sujet et d'envisager le cas.

Packaging :

Conclusion

Tables des illustrations

Annexes

```
graph : [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
stmt_list : [ stmt [ ';' ] stmt_list ]
stmt : node_stmt
     | edge_stmt
     | attr_stmt
     | ID '=' ID
     | subgraph
attr_stmt : (graph | node | edge) attr_list
attr_list : '[' [ a_list ] ']' [ attr_list ]
a_list : ID '=' ID [ (';' | ',') ] [ a_list ]
edge_stmt : (node_id | subgraph) edgeRHS [ attr_list ]
edgeRHS : edgeop (node_id | subgraph) [ edgeRHS ]
node_stmt : node_id [ attr_list ]
node_id : ID [ port ]
port : ':' ID [ ':' compass_pt ]
     | ':' compass_pt
subgraph : [ subgraph [ ID ] ] '{' stmt_list '}'
compass_pt : (n | ne | e | se | s | sw | w | nw | c | _)
```

Figure 8 : Grammaire du langage DOT

(source : http://www.graphviz.org/content/dot-language)

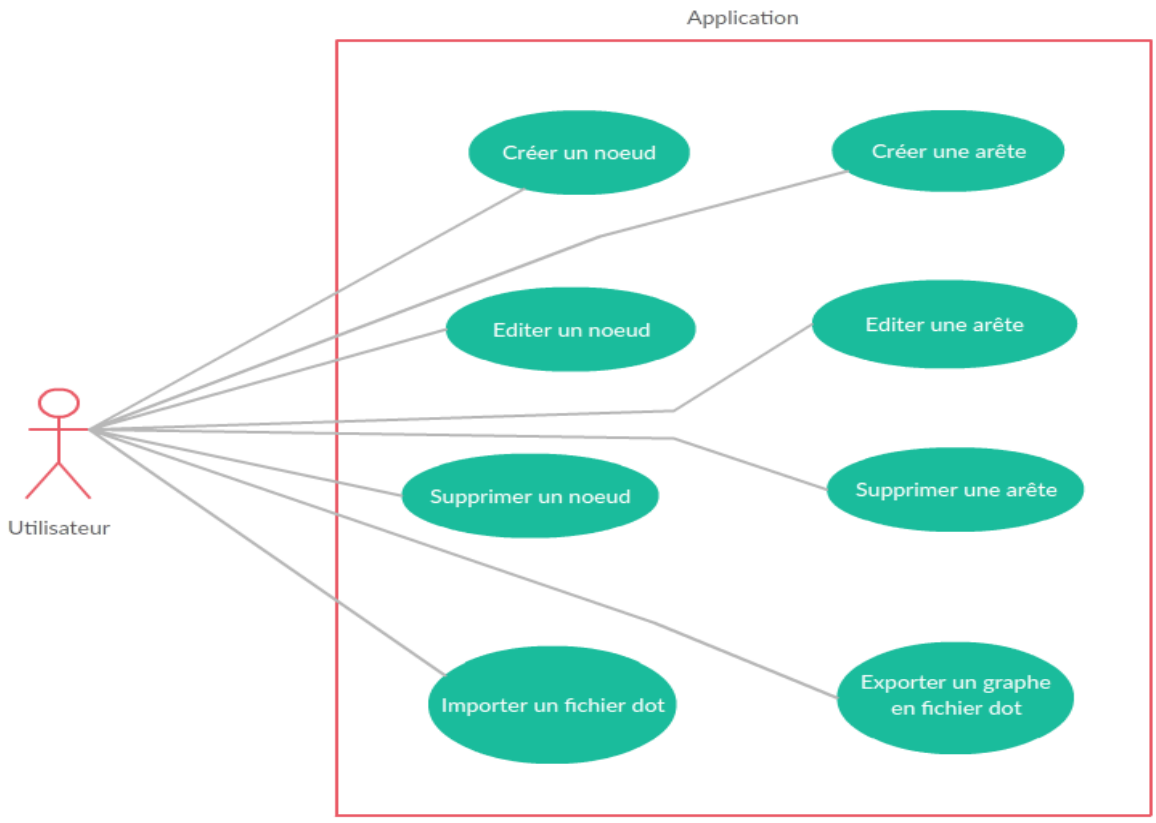

Figure 9: Diagramme de cas d'utilisation

Remarque:

Lorsque l'on parle d'éditer un nœud ou une arête, il s'agit de l'édition d'un de ses attributs (par exemple son label).

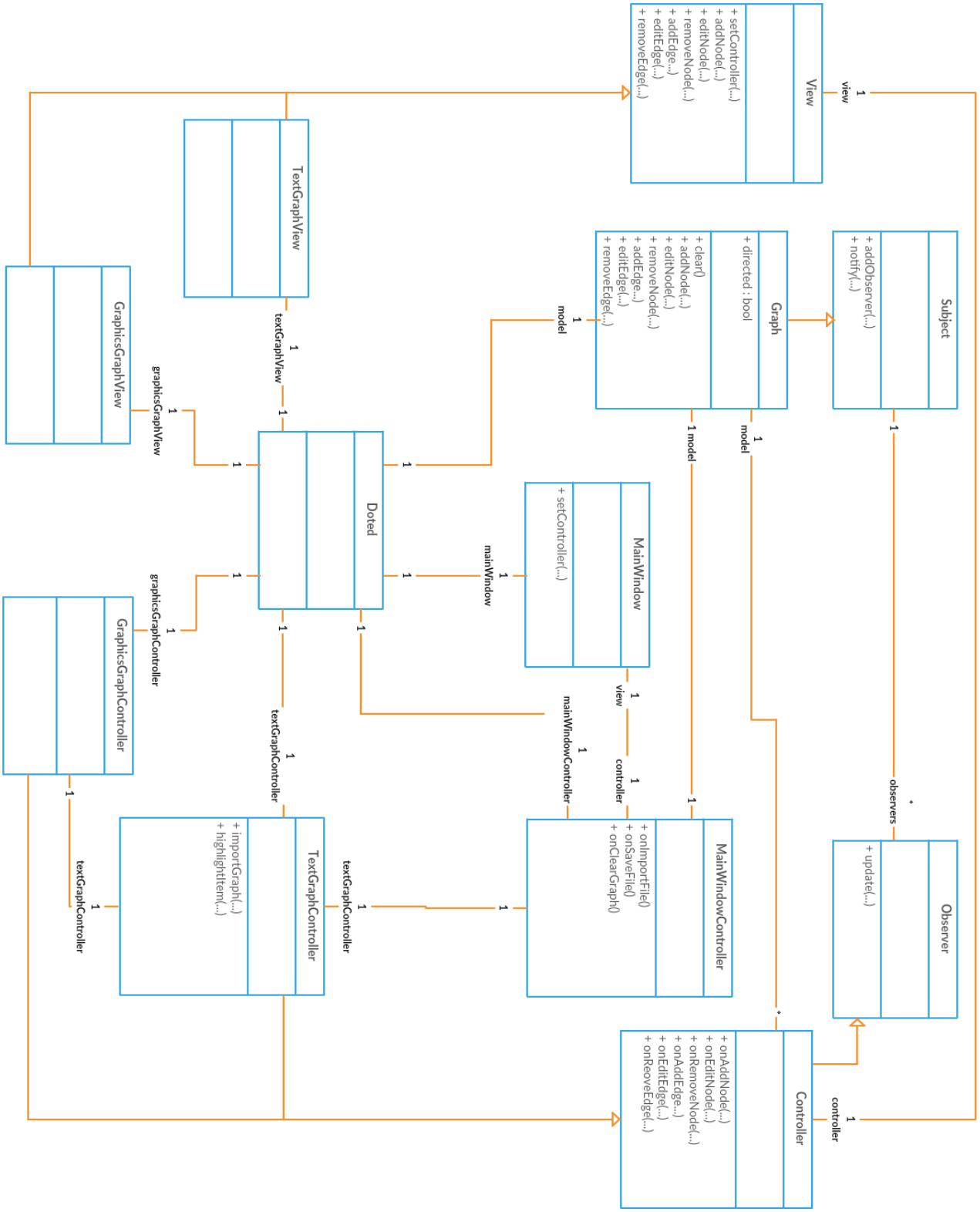

Figure 10 : Diagramme de classe des classes principales de l'application

Remarque :

Le digramme de classes ci-dessus ne considère que les principaux éléments de l'application, c'est à dire qu'il ne contient pas toutes les classes, tous les attributs et toutes les méthodes.

Les noms sur les liens entre classes permettent d'indiquer le sens des références. Par exemple le « Subject » contient plusieurs références sur des « Observer », « MainWindowController » contient une référence de « TextGraphController ».