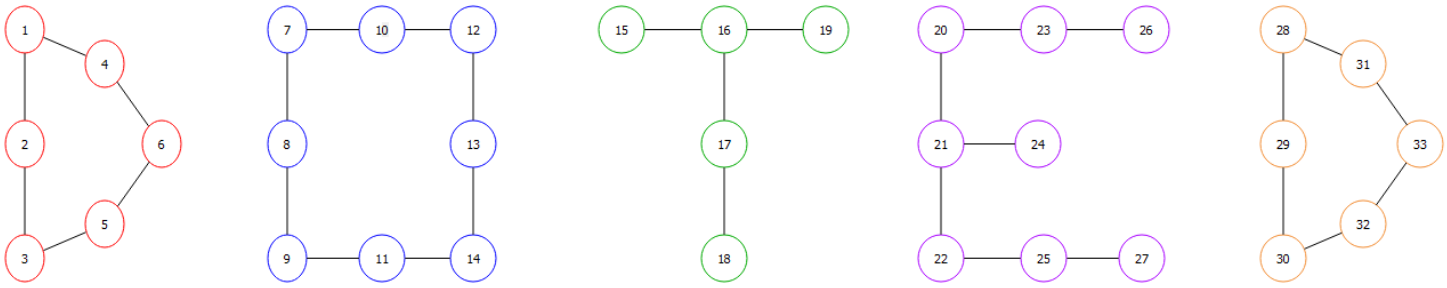


Rapport PSTL

Logiciel d'édition de graphes

Morvan Lassauzay - Victor Nea
Encadrants : Matthieu Dien - Marwan Ghanem

31/05/2016

Sommaire

Introduction

L'utilisation de graphes est fréquente dans la presque totalité des domaines de l'informatique et de l'algorithmique, au point de constituer un impératif journalier pour un grand nombre de chercheurs et professionnels du secteur. Malgré cela, l'édition et la manipulation de ces graphes est encore aujourd'hui souvent lente et peu pratique en raison de la nécessité de recourir à un éditeur de texte pour effectuer ces opérations.

Le format DOT constitue un des principaux formats texte de description de graphes. Notamment utilisé par le logiciel de visualisation GraphViz, il met à disposition de ses utilisateurs une large panoplie d'attributs influant sur le rendu graphique des graphes. Ainsi de manière à faciliter l'utilisation de graphes adoptant ce format, il nous a été demandé de développer un logiciel graphique supportant la plus grande partie du format et permettant l'édition et la manipulation de graphe à la souris.

Afin de présenter l'ensemble du processus de développement de notre logiciel, nous verrons dans un premier temps une phase d'étude du marché des logiciels existants puis de spécification des objectifs à réaliser et des outils utilisés. Nous nous intéresserons ensuite à la conception et à l'implémentation du logiciel en allant de son architecture globale aux détails d'implémentation des interfaces graphiques. Pour finir nous présenterons un exemple d'extension et aborderons succinctement les différents éléments de standardisation permettant la bonne distribution de notre logiciel.

I – Etude préalable

1. 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 de fichier DOT et de sa représentation graphique avec GraphViz :

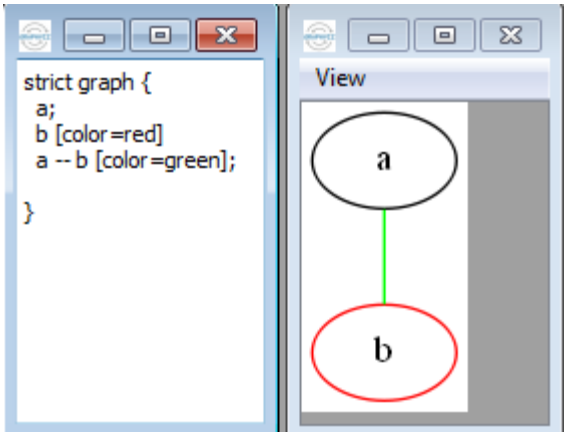


Figure 1 : Représentation graphique d'un fichier DOT

GraphViz est un outil très puissant pour la génération graphique de graphes. Cependant il n'est pas possible de faire des actions comme la création, l'édition ou la suppression d'un élément graphiquement. Le seul moyen consiste donc à modifier le fichier DOT et à régénérer le graphe.

On peut 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 de notre application. Cependant l'édition et la manipulation des graphes à la souris n'est toujours pas au menu.

DotEditor :

DotEditor offre de nombreuses possibilités intéressantes. Il permet tout d'abord d'éditer un graphe via une interface en cliquant sur des boutons et en remplissant des formulaires. On peut ainsi établir un graphe très détaillé sans devoir tout rédiger. 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. Cependant comme pour GraphViz la manipulation à la souris est impossible.

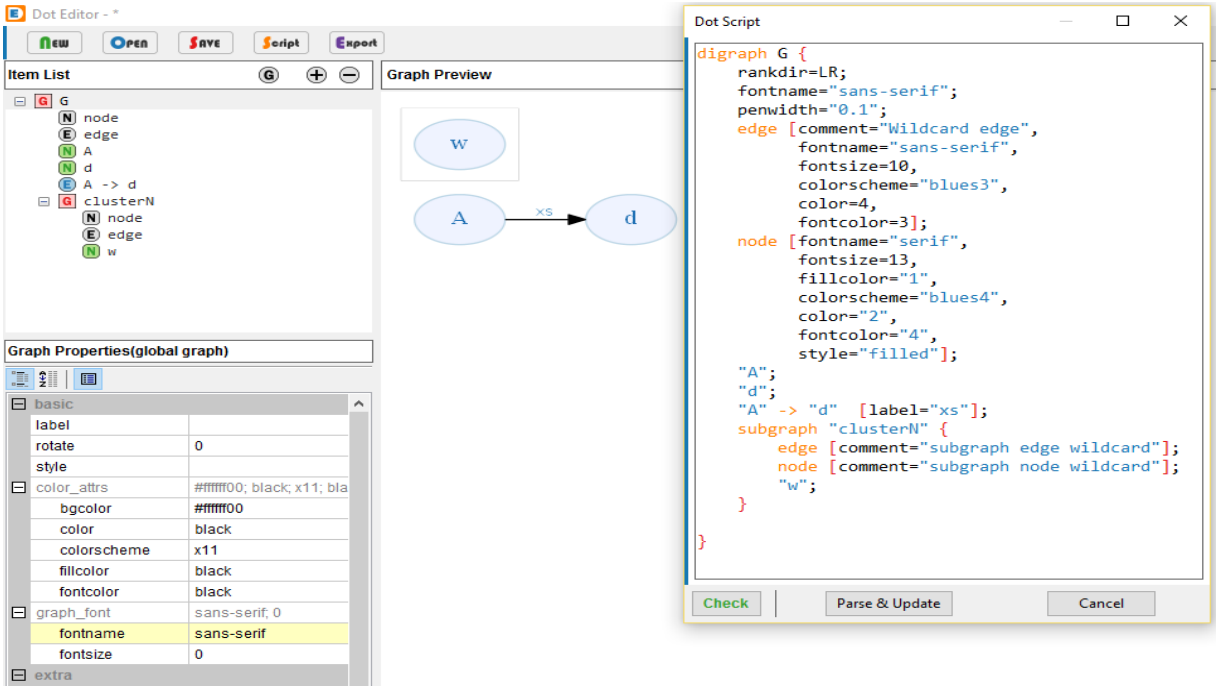


Figure 2 : Capture d'écran du logiciel DotEditor

Autres :

Nous avons trouvé de nombreux autres éditeurs de graphes au format DOT, cependant tous ces logiciels ont été abandonnés en cours de route ou sont tout simplement incomplets ou encore bogués à l'installation. On peut retrouver dans cette liste les logiciels dotty, KGraphEditor, grappa, graphedit.

2. Objectifs du projet

Ce logiciel a pour objectif premier de permettre la manipulation et l'édition à la souris de graphes limités à environ une centaine de nœuds, afin d'en simplifier l'utilisation. En plus de cette fonctionnalité graphique le logiciel doit offrir en parallèle la possibilité d'éditer des graphes à partir d'un texte au format 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 le logiciel 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. De plus il 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, le logiciel se doit de disposer d'une architecture modulaire tout en gardant une organisation claire et efficace.

.

Voici donc la très simple maquette établie à partir des objectifs cités précédemment :

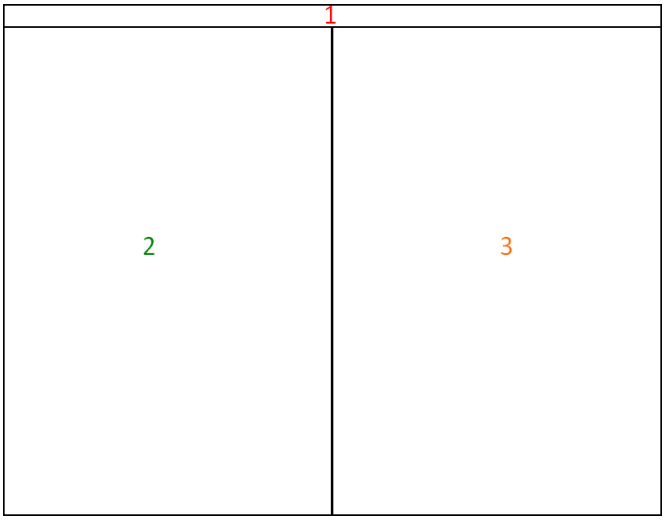


Figure 3 : Maquette du logiciel

1: Barre de menus avec des boutons pour l'import, l'export et la création d'un nouveau graphe.

2: Vue graphique pour afficher, éditer et manipuler le graphe.

3: Vue textuelle pour afficher la description textuelle du graphe au format DOT et permettant également l'édition.

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.

3. Appréhension du format DOT

Le langage DOT permet de définir facilement des nœuds, des arrêtes, des sous-graphes et de leur ajouter des attributs comme un label, une couleur ou encore une forme. On peut dès lors voir apparaître deux phases dans le développement de notre logiciel, 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 diverses attributs et éléments particulier du format. Afin de bien visualiser la syntaxe du format DOT voici un petit exemple d'un graphe utilisant les éléments décrits précédemment :

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

```
        A--C [label=transition];
        D--B;
}
```

Bien que cela ne soit pas obligatoire il nous a été conseillé de considérer que chaque instruction se terminait par un point-virgule (à l'exception des sous-graphes qui utilisent les accolades). On voit également que chaque attribut est défini de la manière suivante : le nom de l'attribut, suivit du caractère égal, suivit de la valeur de l'attribut. On peut donc dès lors imaginer qu'il sera possible de généraliser le traitement des attributs.

4. Outils choisis

Nous avons décidé d'utiliser le langage de programmation Python pour l'implémentation de notre logiciel. 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.
- Le package Pydot offrant un très bon parseur de texte au format Dot et simple d'utilisation.

En ce qui concerne la bibliothèque graphique, nous avons décidé d'utiliser PyQt (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.

II – Implémentation

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

1. Structure et fonctionnement général

1.1 Architecture MVC (Model-View-Controller)

Développer un logiciel graphique complexe était une nouveauté pour nous. Nous avons donc dû étudier et implémenter une architecture MVC 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 précises, 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 différentes 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. 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 action de l'utilisateur.

Le diagramme de classes ci-dessous permet de disposer d'une représentation explicite de l'architecture obtenue.

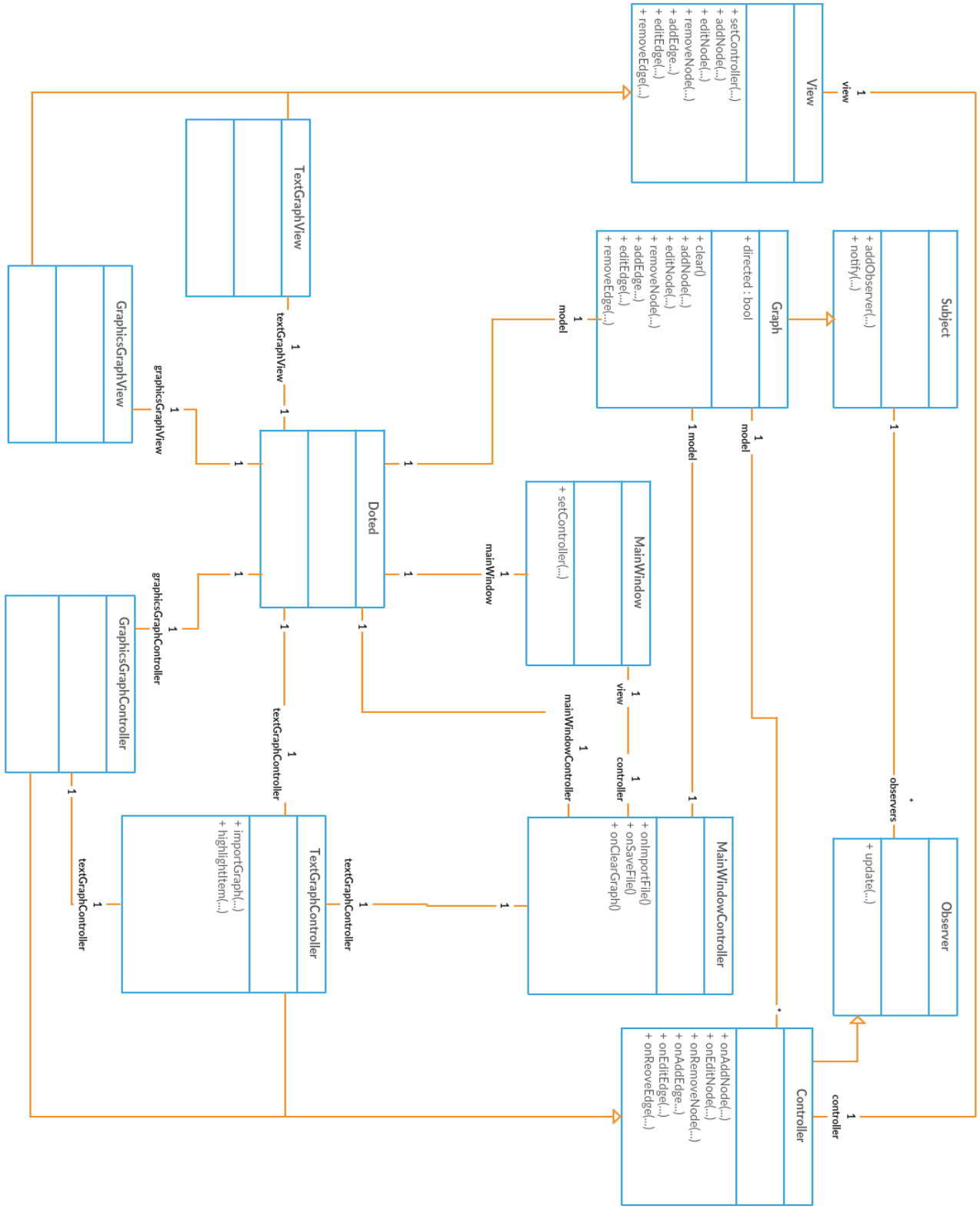


Figure 4 : Diagramme de classe des classes principales du logiciel

Le digramme de classes ci-dessus ne considère que les principaux éléments de l'application, il ne contient pas toutes les classes, ni tous les attributs, ni 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 ». La classe « Graph » représente le modèle, tandis que « Doted » est la classe principale.

1.2 Modèle et mises à jour

Pour bien comprendre le fonctionnement du logiciel 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 actuelle, 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 le logiciel, 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 elles ne sont constituées que d'un nœud source et d'un nœud destination. Le format Dot n'exclut pas les arêtes 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 étant donné que les arêtes ne disposent 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'identifiant du nœud ou de l'arête concerné ainsi qu'un dictionnaire contenant les informations utiles (label, position, nœud source, etc.). Une clé unique pour chaque type d'information (label, position, nœud source, etc.) est stockée dans des énumérations. Cela nous permet de récupérer ces informations tout au long de la chaine de mise à jour 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 :

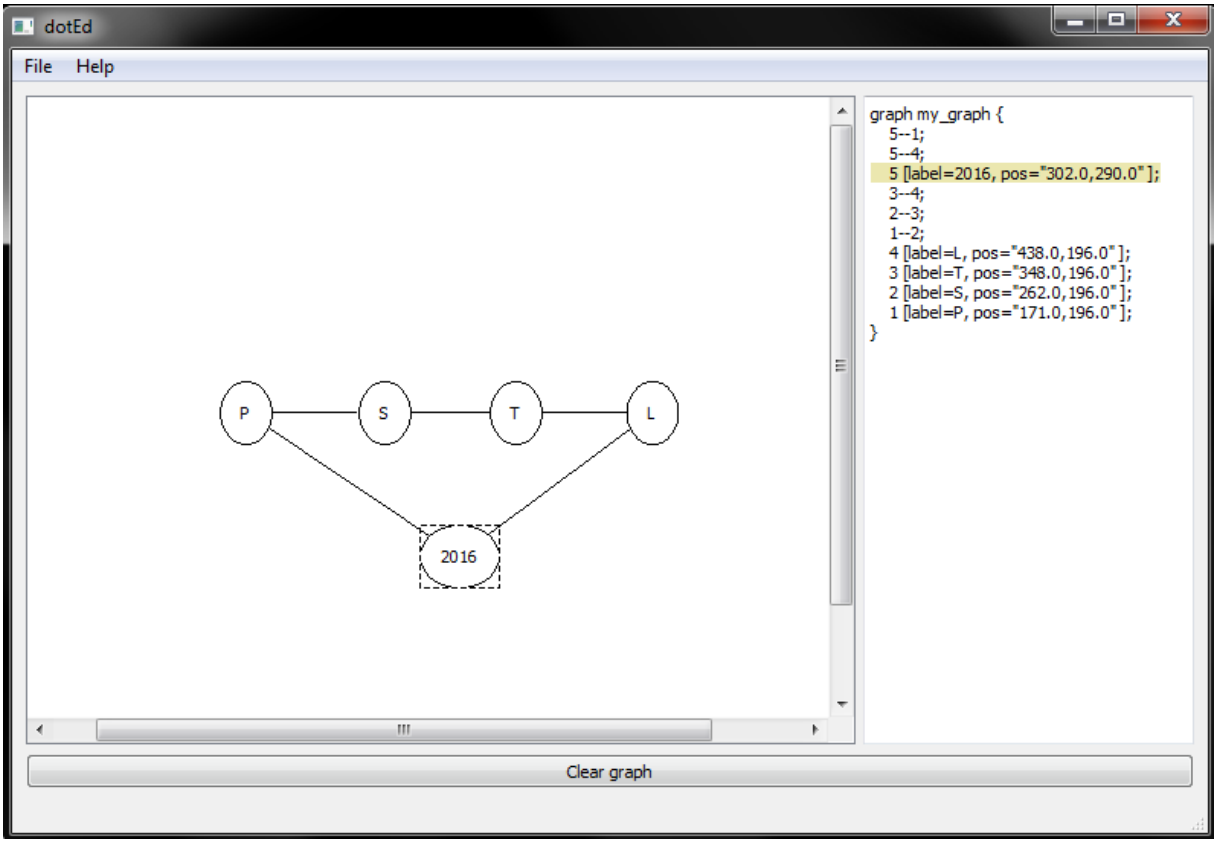


Figure 5 : Aperçu de l'application

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 qui seront décrites plus tard. 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 PyQt 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 fournie par PyQt 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'outil 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 Fonctionnement de la vue graphique et implémentation

La bibliothèque graphique PyQt nous offre la classe QGraphicsView qui est un canevas vectoriel. Autrement dit c'est une surface qui permet de dessiner des éléments graphiques et d'y appliquer des transformations (translation, rotation, mise à l'échelle, etc.). Elle nous offre donc une grande flexibilité pour la manipulation d'éléments graphiques (noeuds, arêtes, etc.) et correspond donc exactement à nos attentes.
La classe QGraphicsView fonctionne en complémentarité avec la classe QGraphicsScene. Cette scène va contenir tous les éléments graphiques, à savoir dans notre cas les noeuds et les arêtes. L'intérêt de séparer la vue et la scène est de permettre à plusieurs vues d'observer une même scène mais avec différentes transformations. Par exemple on peut avoir une vue qui est zoomée et une autre qui ne l'est pas. Il faut donc comprendre que c'est la vue qui dessine et applique les transformations.

Pour pouvoir interagir avec le logiciel et plus précisément avec cette vue graphique, PyQt permet 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.

Implémentation de la vue graphique GraphicsGraphView :

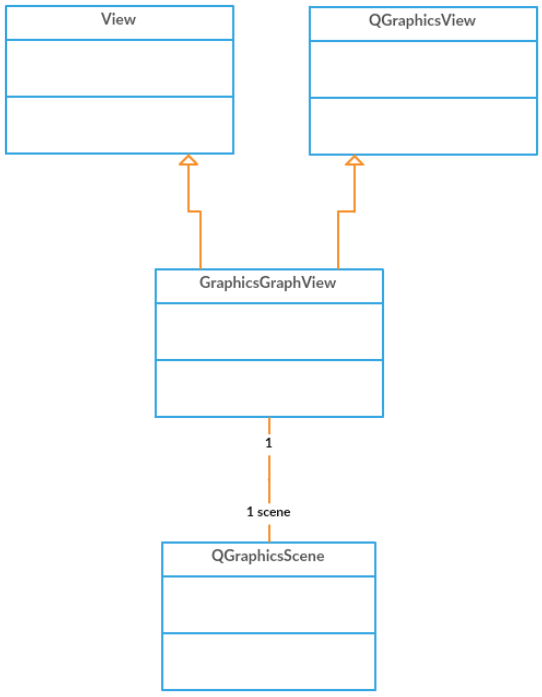


Figure 6 : Diagramme de classes de GraphicsGraphView

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 qui va contenir l'ensemble de nos éléments graphiques.

2.2.2 Implémentation des nœuds graphiques

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

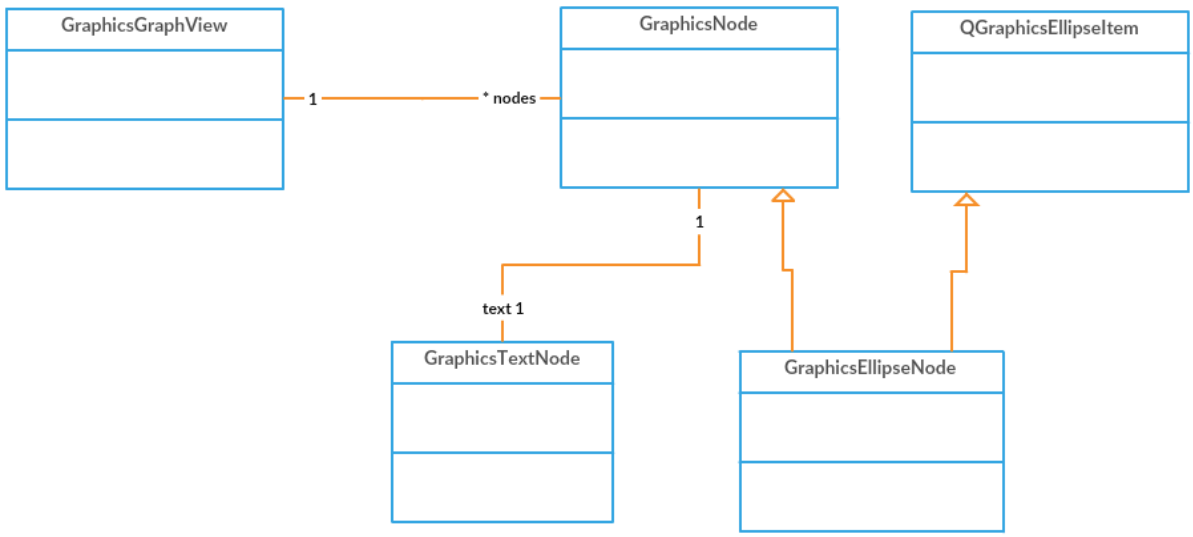


Figure 7 : Diagramme de classes de GraphicsGraphView avec nœuds

Pour l'implémentation 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 Implémentation des arêtes graphiques

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

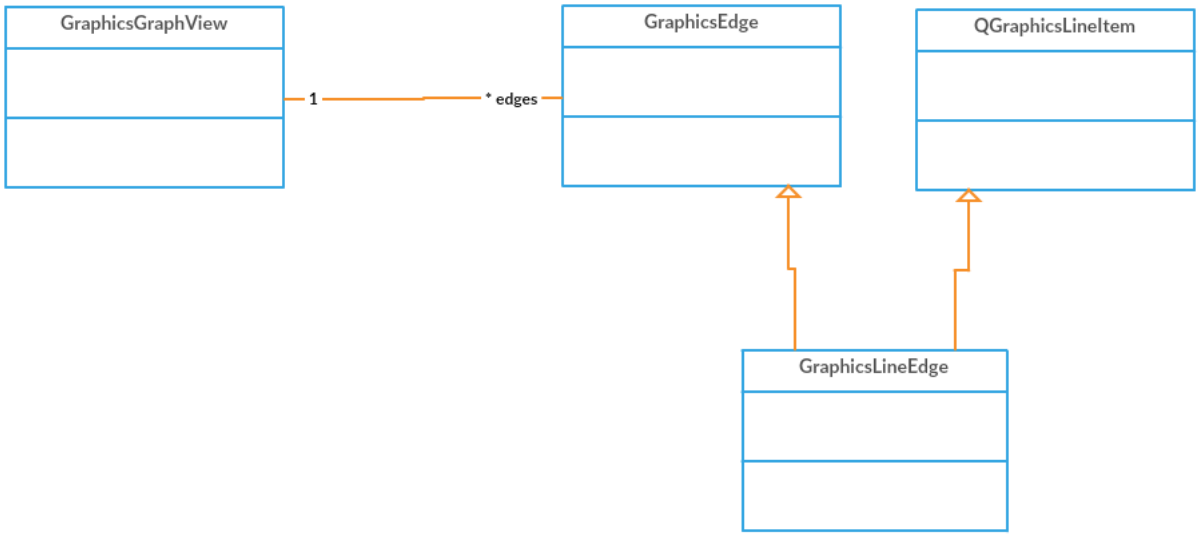


Figure 8 : Diagramme de classes de GraphicsGraphView avec arêtes

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.

2.3 La vue textuelle : TextGraphView

Cette vue ne dispose pas réellement d'éléments graphiques, elle se contente d'hériter de la classe QTextEdit permettant d'afficher et d'éditer un texte. Cependant elle constitue sans doute la partie la plus délicate de l'application.

2.3.1 Edition du graphe via le texte

Pour ce cas nous avons rencontré deux principales difficultés. La première est la conservation du texte dans un format DOT correct. En effet la vue s'utilisant comme un éditeur de texte classique, l'utilisateur dispose d'une grande liberté pour effectuer ses modifications et il est alors difficile de s'assurer que ce dernier à laisser le texte dans un état valide. La seconde difficulté rencontrée se résume par la question suivante : Comment pouvons-nous faire pour savoir exactement quels éléments ont été modifiés par l'utilisateur ?

Pour résoudre ce problème d'édition du graphe à partir du texte, nous avons eu recours au parseur Pydot. 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 en mémoire dans notre vue les anciens éléments présents dans le graphe. Ainsi à chaque fois que

l'utilisateur effectue une modification sur le texte, nous commençons par parser le texte puis nous comparons 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 posent 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 clique 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 *parsing*. 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 (visibles dans le texte sans engendrer d'effets de bord), 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 (j'ai pas compris lol), 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 Modification du texte sur ordre du modèle

Certainement le point le plus ardu de notre logiciel. 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 une copie du modèle contenue dans la vue puis nous réécrivions le texte à partir de cette copie. Cependant cette technique ne permettait pas de garder la forme que l'utilisateur avait donné au texte.

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 lorsqu'une modification survient 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 eu recours à Pydot pour effectuer le processus suivant. Dans un

premier temps nous découpons notre texte sur chaque point-virgule afin de récupérer la liste ordonnée des instructions décrivant le graphe. Ensuite nous parsons ces instructions une à une dans l'ordre du document jusqu'à rencontrer l'instruction correspondant au nœud ou à l'arête ayant subi une modification. Pour chaque instruction parcourue 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 suivants, n représentant la taille de l'instruction. S'il s'agit d'une modification d'attribut, nous recherchons dans l'instruction du nœud la place exacte (en nombre de caractère) de l'attribut modifié à l'aide d'une expression régulière. Une fois cela effectué nous récupérons l'ancienne valeur de l'attribut à partir d'une copie des éléments du graphe stockée en mémoire dans la vue. Nous obtenons alors la taille n de cette valeur puis nous supprimons le bon nombre de caractères du texte en fonction de cette taille n. Il ne nous reste ensuite plus qu'à recopier la nouvelle valeur de l'attribut à la bonne place. Résumons les différentes étapes décrites précédemment dans le cas de la modification du label d'un nœud N:

1- Découpe du texte afin d'obtenir la liste ordonnée des instructions du graphe.
2- Recherche du caractère auquel commence l'instruction correspondant à N.
3- Recherche du caractère où se trouve le label de N.
4- Récupération de la taille de l'ancienne valeur du label et suppression du nombre de caractères correspondant.
5- Ecriture de la nouvelle valeur du label de N.

Dans le cas d'un ajout, le nœud ou l'arête est simplement écrit en tête de la description du graphe. En effet savoir à quel endroit l'utilisateur souhaite placer cet élément dans le texte est impossible.

Vous avez certainement remarqué sur les différents visuels du logiciel 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 attribut de 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

Il existe plusieurs types d'extension mais ici nous allons nous intéresser à la gestion d'un nouvel attribut DOT pour les nœuds. Nous avons choisi de traiter l'attribut DOT color.

1.1 Choix de l'attribut

Tout d'abord, il faut se rendre à cette adresse pour voir la liste des attributs DOT : http://www.graphviz.org/doc/info/attrs.html. Une fois sur le site, on distingue un tableau avec plusieurs colonnes. Nous allons seulement nous intéressés aux colonnes suivantes:

- Name : Nom de l'attribut DOT.
- Used by : Peut contenir les lettres E pour edges (arêtes), N pour nodes (nœuds), G pour root graph (graphe principal), S pour subgraphs (sous-graphes) et C pour cluster graphs (groupe de graphes). Cela signifie qu'un attribut peut être utilisé par un ou plusieurs de ces éléments.
- Type: Type de l'attribut DOT.
- Default: Valeur par défaut de l'attribut.

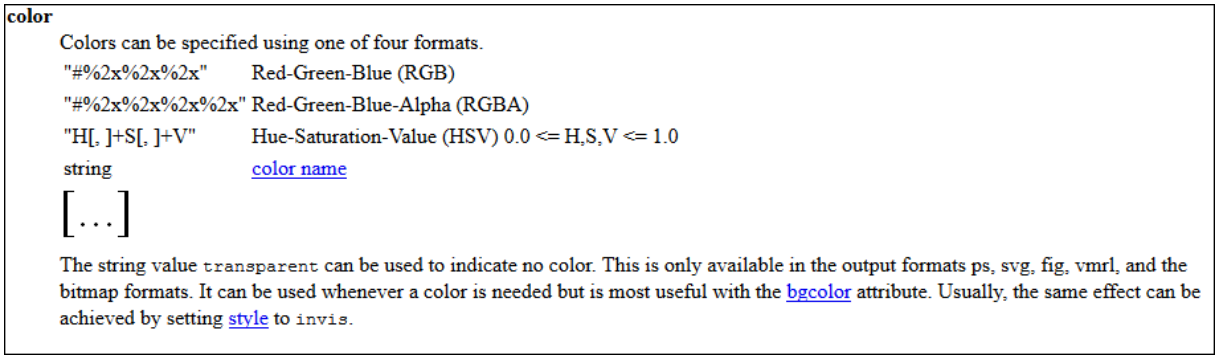
Name	Used By	Type	Default
color	ENC	color colorList	black

Figure 9 : Attribut DOT color

Dans notre cas, il faut donc trouver l'attribut color dans la colonne Name. Une fois que l'on a trouvé cette ligne, il faut regarder si la colonne Used by contient bien la lettre N pour nodes (nœuds). On peut voir que c'est bien le cas sur l'image ci-dessus. Sous la colonne Type, on peut voir que l'attribut color peut prendre deux types différents: les types color et colorList. Nous ne traiterons pas le type colorList car il dépend d'autres attributs DOT. Il aurait donc fallut d'abord ajouter les attributs liés à ce type. Si un attribut color est de ce type, il faudra tout simplement mettre comme couleur par défaut black dans la vue graphique mais laisser la valeur telle quelle dans la vue textuelle.

1.2 Identification des valeurs possibles d'un attribut

Revenons en au type color : en cliquant sur le lien color (sous la colonne Type), nous pouvons voir comment celui-ci est défini, c'est à dire quelles sont les différentes valeurs qu'il peut prendre.



color

Colors can be specified using one of four formats.

"#%2x%2x%2x"	Red-Green-Blue (RGB)
"#%2x%2x%2x%2x"	Red-Green-Blue-Alpha (RGBA)
"H[,]+S[,]+V"	Hue-Saturation-Value (HSV) 0.0 <= H,S,V <= 1.0
string	color name

[. . .]

The string value `transparent` can be used to indicate no color. This is only available in the output formats ps, svg, fig, vmrl, and the bitmap formats. It can be used whenever a color is needed but is most useful with the bgcolor attribute. Usually, the same effect can be achieved by setting style to `invis`.

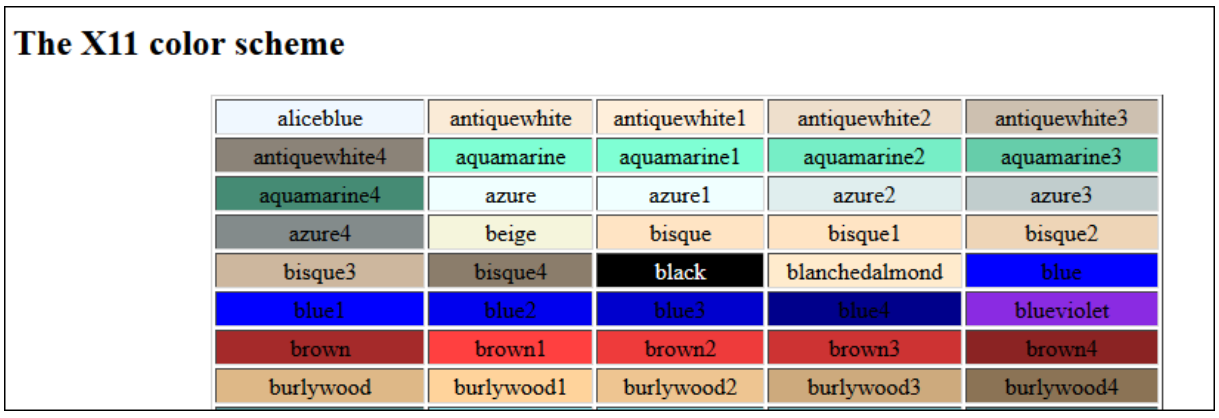
Figure 10 : Description du type color

Comme on peut le voir sur l'image ci-dessus, la type color peut prendre comme valeurs les formats suivants :

- RGB : Red Green Blue.
- RGBA : Red Green Blue Alpha.
- HSV : Hue Saturation Value.
- string (color name) : Un nom de couleur prédéfini.
- transparent: Le contour du nœud devient invisible.

Comme on s'en doute, nous allons plus tard devoir créer une fonction qui vérifiera si la valeur du type est valide ou non. Pour les cas RGB, RGBA, HSV et transparent il sera très facile de vérifier si la valeur est valide ou non et il sera également facile d'obtenir la couleur correspondante grâce à la classe QColor de PyQt.

Cependant le cas string (color name) car il faut trouver un moyen de convertir la couleur blue en QColor. Pour cela il faut cliquer sur le lien color name qui nous affiche toutes les valeurs color name possibles:



The X11 color scheme

aliceblue	antiquewhite	antiquewhite1	antiquewhite2	antiquewhite3
antiquewhite4	aquamarine	aquamarine1	aquamarine2	aquamarine3
aquamarine4	azure	azure1	azure2	azure3
azure4	beige	bisque	bisque1	bisque2
bisque3	bisque4	black	blanchedalmond	blue
blue1	blue2	blue3	blue4	blueviolet
brown	brown1	brown2	brown3	brown4
burlywood	burlywood1	burlywood2	burlywood3	burlywood4

Figure 11 : Liste des color name

Nous obtenons donc toutes les color name possible et en examinant le code source de la page web on peut constater que pour chaque couleur, on peut obtenir sa couleur RGB correspondante. Par exemple pour le code source de la couleur aliceblue on a : aliceblue.

Il est donc possible d'obtenir un fichier texte de correspondance des couleurs suivant ce modèle:
aliceblue #f0f8ff
antiquewhite #faebd7
... ...

Remarque :
La création de ce fichier texte peut se faire simplement en JavaScript à l'aide de sélecteurs web. Cependant la démarche complète ne sera pas décrite ici car il peut exister d'autres outils que JavaScript pour extraire ces données dans un fichier texte.

1.4 Implémentation

Cette étape consiste à implanter le code gérant ce nouvel attribut color.

Il faut d'abord commencer par rajouter dans la liste des énumérations de NodeDotAttrs le nouvel attribut color.

Ensuite il faut créer une classe NodeDotColorUtils qui va contenir ces 3 méthodes (statiques) :

- formatColor(color) : Met en forme la valeur color pour qu'il soit au bon format.
- getColor(colorAttr) : Retourne la QColor en fonction de la valeur de l'attribut.
- isColorValid(colorAttr) : Vérifie si la valeur de l'attribut color est valide à l'aide d'expressions régulières et du fichier texte que l'on a obtenu précédemment.

Après cela, il faut rajouter le fait qu'il faille vérifier la validité de l'attribut color pour les nœuds. Pour cela il faut créer une nouvelle classe DotAttrsUtils qui hérite de l'ancienne. Il faut ensuite redéfinir la méthode checkNodeAttrsForm(...) en vérifiant la valeur de l'attribut color à l'aide de la méthode isColorValid(colorAttr) de la classe NodeDotColorUtils.

Enfin, il faut créer une nouvelle classe GraphicsNode qui hérite de l'ancienne et il faudra :

- Rajouter dans le constructeur l'action « Edit color » dans le menu contextuel.
- Rajouter le callback onEditNode qui permet de sélectionner une nouvelle couleur lors de la sélection de Edit color. Cette méthode utilisera la méthode formatColor(color) de la classe NodeDotColorUtils pour formater la couleur.
- Rajouter la méthode editColor qui permet de mettre à jour la couleur lors d'une mise à jour extérieure. Cette méthode utilisera la méthode getColor(colorAttr) de la classe NodeDotColorUtils pour obtenir la nouvelle QColor.

Remarque :

On pourrait croire qu'il faudrait également retoucher à la classe Node qui est la classe modèle pour les nœuds. Cependant il n'est pas nécessaire d'y retoucher car elle a été pensée pour être générique lors de l'ajout d'un nouvel attribut comme il a été dit plus tôt dans le rapport.

Pour finir il faut simplement redéfinir les classes qui dépendent de ces nouvelles modifications avec les nouveaux import, c'est à dire utiliser les nouvelles classes et au lieu des anciennes. Il faut également définir un nouveau fichier main.py pour une nouvelle version de l'application.

1.4 Résumé

Pour résumer, il faut d'abord choisir un attribut DOT d'un élément DOT sur le site officiel, identifier les différents valeurs possibles que peut prendre cet attribut et enfin implémenter le code. Il est donc simple d'ajouter la gestion d'un nouvel attribut DOT en suivant ces différentes étapes et on peut noter que le projet est bien modulaire. On peut estimer qu'il est nécessaire d'écrire entre 50 et 300 lignes de code (sans compter les commentaires, les imports, etc.) pour ajouter une extension d'un attribut DOT. Ce qui prend le plus de temps pour développer une extension de ce type et de bien vérifier toutes les valeurs possibles de l'attribut.

2. Standardisation

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. Voici quelques exemples de conventions : espaces à la place de tabulations, 80 caractères par ligne au maximum, etc.

Pour notre application nous avons choisi d'utiliser la licence GNU GPL (GNU General Public License) qui offre de nombreux avantages. Un 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 de nouveaux attributs ou d'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 dans les faits cela soit difficilement applicable. Bien entendu avant d'utiliser cette licence nous avons vérifié sa comptabilité avec les licences des différents outils utilisés par notre programme : MIT pour Pydot et GNU GPL pour PyQt5.

Python dispose d'un dépôt de package officiel nommé PyPI. C'est à partir de ce dépôt que la commande d'installation de package python « pip » effectue les téléchargements. Afin de simplifier au

maximum la distribution de notre logiciel nous avons respecté le format de package recommandé par PyPI. Ainsi il est possible d'installer très rapidement notre logiciel par la commande « pip install doted » puis de l'exécuter grâce à la commande « python doted ». Afin de fonctionner notre logiciel nécessite cependant l'installation préalable de la bibliothèque PyQt5.

Notre projet est disponible à l'adresse suivante : https://github.com/vnea/dotEd. Il dispose d'une section Wiki (https://github.com/vnea/dotEd/wiki) qui décrit l'organisation des packages du projet et qui explique également comment rajouter une extension (comme dans le rapport). Cela permettra aux personnes intéressées de pouvoir reprendre assez facilement notre projet.

Conclusion

Exemples d'extension : système de positions relatives des noeuds par rapport aux autres, extension de la grille avec les noeuds qui sont attirés, graphes dirigés, nouveaux attributs DOT, différents format d'exportation, gestion de copier/coller/couper dans la vue graphique, etc.

Sinon j'avais noté ça par rapport à la présentation (enfin les conseils des profs) :
Etre plus positif sur la conclusion :

=> Logiciel d'édition de graphes qui fonctionne (import/export)

=> Intégration dans le langage python

=> gestion partielle des attributs DOT -> gestion partielle GRAPHIQUE des attributs DOT (comme sur la vue texte ils sont gérés mais pas traités).