ESTEBE Anthony

Logiciel pédagogique pour la compréhension d'algorithmes de graphes

supervisé par Mr Olivier Cogis

Remerciements

Pour le bon déroulement de ce projet, nous avons pris contact avec plusieurs personnes, pour cela et avant toute chose nous tenons donc à les remercier.

- Mr Olivier Cogis, notre tuteur de projet, qui nous a proposé ce sujet, mais qui a surtout été un très bon tuteur car il a réussi à nous aiguiller, tout en nous laissant une certaine liberté. De plus, nous avons été régulièrement amenés à avoir des discussions fortes enrichissantes, qu'elles concernent le projet ou non.
- Madalina Croitoru, mon enseignante de programmation, qui a montré un certain intérêt concernant le projet, et qui nous a donné diverses pistes concernant la représentation graphique du logiciel.
- Mr Xavier Palleja, professeur d'UML, avec qui, nous avons eu plusieurs discussions à propos de l'architecture générale du logiciel, ainsi que de différentes portions du logiciel.
- D'autre professeurs sont également intervenus lors de problèmes rencontrés au sein du projet et qui ont pu nous aider du mieux possible, comme Mr Artignan ou encore Mr Palaysi.

Table des matières

Table des figures

Glossaire

- **Graphe** : Un graphe G est un couple $G = (X, E)$ constitué d'un ensemble X non vide et fini, et d'un ensemble E de paires d'éléments de X.
- **Nœud** : Élément appartenant à l'ensemble X.
- **Arc** : Élément appartenant à l'ensemble E.
- **Algorithme** : Ensemble d'instructions permettant d'effectuer une tâche.
- **C++** : Langage de programmation permettant la programmation objet.
- **XML** : (Extensible Markup Language) Sert essentiellement à stocker/transférer des données de type texte structurées en champs arborescents.
- **Pattern** : Concept de génie logiciel destiné à résoudre les problèmes récurrents en programmation.
- **M.V.C.** : (Modèle, Vue, Contrôleur) Scinde le logiciel en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.
- **U.M.L.** : (Unified Modeling Language) Langage de modélisation graphique à base de pictogrammes.
- **Classe** : Propriétés communes à un ensemble d'objets.
- **Interface** : Classe ne contenant aucun code.
- **Thread** : Exécutions d'instruction semblant se dérouler en parallèle pour l'utilisateur.
- **Sérialisation** : Processus visant à encoder l'état d'une information qui est en mémoire.
- **Greffon** : Logiciel qui complète un logiciel hôte pour lui apporter de nouvelles fonctionnalités.
- **Package** : Utilisé afin de regrouper un ensemble de classes ou d'interfaces.
- **F.I.F.O.** : (First in first out) Méthode de traitement de donnée où l'on récupère la première valeur.
- **L.I.F.O.** : (Last in first out) Méthode de traitement de donnée où l'on récupère la dernière valeur.

1 Introduction

Ce projet a été réalisé dans le cadre de la Licence Professionnelle Assistant Chef de Projet Informatique. Ce projet, proposé par Mr Olivier Cogis, a pour but d'aider à la compréhension d'algorithmes de graphes. Un graphe est une structure de donnée permettant de relier des éléments deux à deux. Il possède plusieurs éléments appelés des nœuds, et plusieurs liens appelés des arcs. Cette structure de donnée très pauvre en informations peut être utilisée dans une multitude de problèmes. En effet, un graphe peut servir à représenter un simple réseau d'amis, une carte routière, le réseau internet et bien d'autres.

Un graphe peut vite être très important, avec des centaines voir des milliers d'arcs, et lorsqu'un graphe de la sorte se présente, il est très difficile de trouver les informations recherchées. De ce fait, nous utilisons des algorithmes qui permettent de résoudre ces problèmes de façon automatique et plus rapidement que si nous cherchions les solutions nous-même. Ces algorithmes bien que très importants, peuvent facilement être compliqués à comprendre. Ce logiciel pourra donc s'avérer utile pour voir le déroulement des algorithmes, ainsi que pour interagir avec ceux-ci en leurs indiquant, par exemple, quels nœuds choisir.

Ce logiciel pourra, de plus, servir par exemple de débogueur d'algorithme de graphes. En effet, il sera possible de suivre pas à pas l'avancement de l'algorithme et donc ceci permettrait de voir à quel endroit l'algorithme ne fait pas ce qu'il devrait faire. Enfin, ce logiciel pourra s'avérer pratique lors de cours, de présentations ou autre.

2 Analyse du besoin

Afin de réaliser un projet au plus proche des attentes du tuteur de projet, il a été nécessaire de faire une analyse du sujet de manière à pouvoir en discuter avec lui et de voir si nous avions bien compris le sujet.

2.1 Principales fonctionnalités

Nous pourrons visualiser le graphe et le déroulement de l'algorithme à l'aide d'une interface graphique. Cette interface pourra également proposer différents algorithmes de graphes tels que trouver les descendants d'un sommet d'un graphe ou bien calculer la numérotation compatible d'un graphe sans circuit.

L'utilisateur aura la possibilité de créer, sauvegarder et modifier des graphes à sa guise, ou pourra choisir un graphe parmi une bibliothèque de graphes.

L'utilisateur pourra lors du déroulement d'un algorithme et selon celui-ci, interagir pour modifier l'exécution de l'algorithme (comme par exemple choisir un sommet).

Nous devrons également fournir une bibliothèque d'algorithmes, ainsi qu'une documentation pour le développement d'autres algorithmes éventuels.

Le logiciel pourra être évolutif et sera donc facilement modifiable. Il supportera l'ajout de nouveaux algorithmes et de nouvelles structures de graphe aisément.

2.2 Diagramme UML

2.2.1 Diagramme de use case

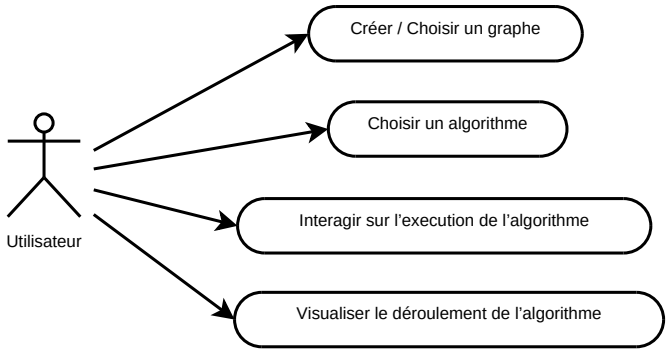

Fig. 1 – Diagramme de Use case

Scénario nominal : L'utilisateur choisit un graphe parmi la liste des graphes disponibles. Le logiciel charge le graphe et affiche à l'écran la visualisation de celui-ci. L'utilisateur choisit un algorithme parmi une bibliothèque d'algorithme puis demande son lancement. A chaque étape

importante de l'algorithme, le logiciel affichera l'état du graphe et ainsi l'utilisateur pourra voir le déroulement de l'algorithme et, suivant les algorithmes, pourra l'orienter en choisissant par exemple un sommet.

2.2.2 Diagramme de classe

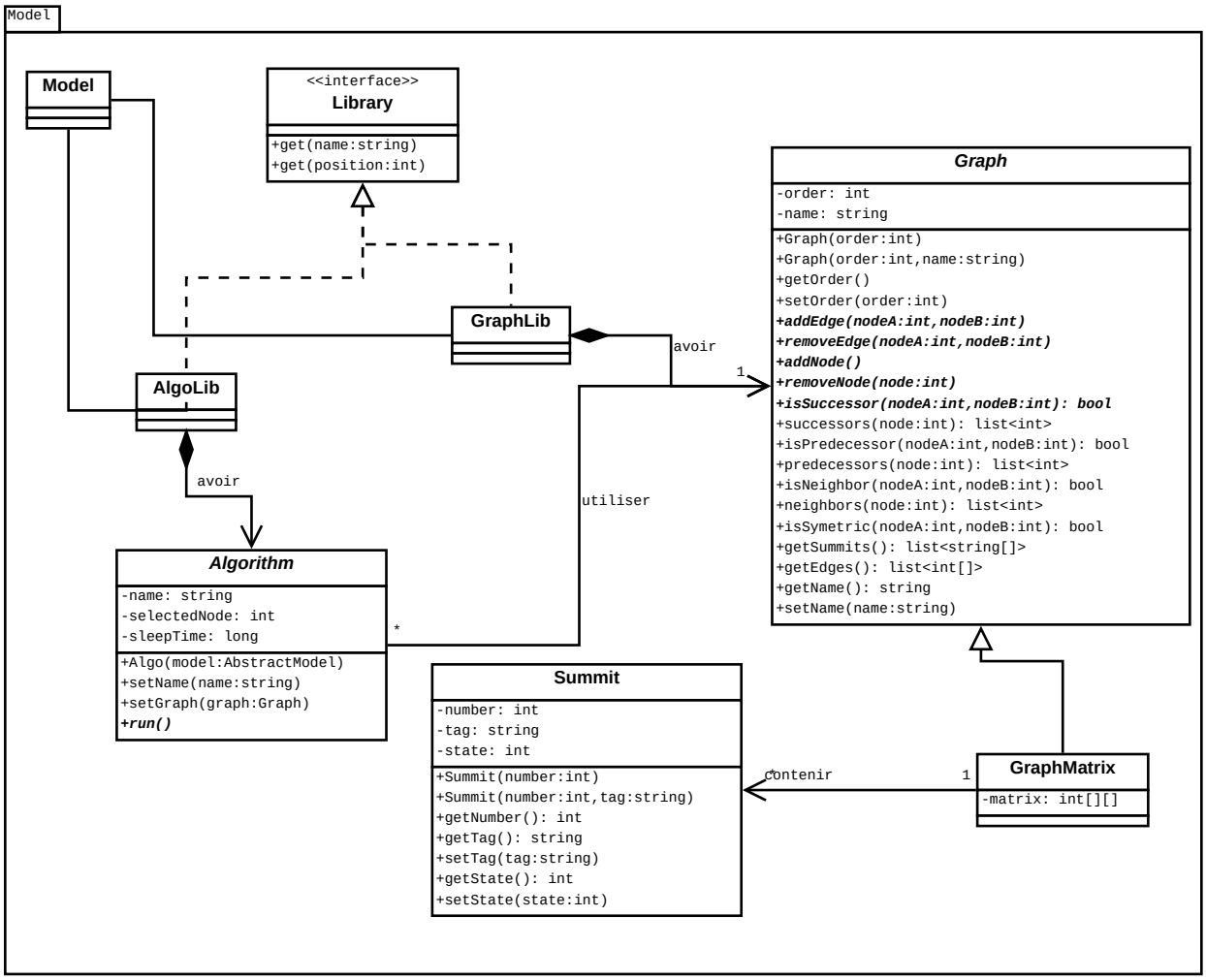

FIG. 2 – Diagramme de classe d'analyse

2.3 Contraintes non fonctionnelles

Très peu de contraintes non fonctionnelles ont été relevées. En effet, le sujet était assez ouvert, nous n'avions aucune contrainte de technologies, ou bien de plateformes. Les quelques contraintes que nous avons pu recenser sont :

- Le développement du logiciel à l'aide du pattern Modèle Vue Contrôleur (M.V.C.) afin de donner une certaine flexibilité à l'architecture du logiciel.
- Le logiciel devra manifester un souci d'ouverture afin de permettre de possibles évolutions.

– Après notre premier rendez-vous avec notre tuteur il a été décidé que ce projet sera développé en C++.

2.4 Outils utilisés

Afin que le projet se déroule au mieux, divers outils ont été utilisés, qu'ils soient pour l'analyse du projet, son élaboration, ou bien sa gestion. Tout d'abord, pour la partie analyse et conception, il a été utile de pouvoir faire différents diagrammes UML, pour cela deux outils ont été utilisé :

– ArgoUML : Logicel libre permettant la création de digramme UML respectant la norme UML1.4. Cet outil permet également la génération de code à partir de diagrammes et ce dans différent langages tels que le C++, le Java, le PHP ...
http ://argouml.tigris.org/

– Dia : Logiciel libre de création de diagrammes de tous types. Il peut également permettre l'édition de diagramme UML, cependant celui-ci est moins abouti qu'ArgoUML, mais en contrepartie, permet des représentations beaucoup plus lisibles des diagrammes réalisés.
http ://live.gnome.org/Dia/

Par la suite, pour le développement nous avons utilisé des environnement de développement intégré (I.D.E.) ainsi que des débogueurs :

– Eclipse 3.5 Galileo : I.D.E. libre utilisé pour le développement du cœur du logiciel (c'est-à-dire la partie modèle principalement).
http ://www.eclipse.org/

– Qt4 : Bibliothèque graphique orientée objet et développée en C++ permettant de concevoir des interfaces graphiques à l'aide de nombreux outils déjà intégrés à la librairie. Cette bibliothèque graphique est très intéressante de part sa portabilité, en effet portable sur des systèmes Unix (Linux et Unix embarqué), Windows Mac OS mais aussi car c'est une bibliothèque qui a été adaptée dans de nombreux langages (C++, Java, Python, C#).
http ://qt.nokia.com/

– QtCreator : I.D.E. développé par Qt Development Frameworks permettant de coder plus aisément des interfaces graphiques se basant sur la bibliothèque graphique Qt. Cet I.D.E. fournit différentes fonctionnalités telles que la création des fenêtres de façon graphique, l'aide associée à la bibliothèque graphique, etc...
http ://qt.nokia.com/products/appdev/developer-tools

– Valgrind : Débogueur en mode console permettant de faire du profilage de code et de pouvoir repérer les fuites de mémoires.

http ://valgrind.org/

– QtCreator Débogueur : Outils intégré à QtCreator, permettant le débogue d'un programme et plus particulièrement des interfaces graphiques.
http ://qt.nokia.com/products/appdev/developer-tools

Enfin, pour l'organisation du projet, nous avions prévu d'utiliser un outil tel qu'une forge gérant le versionnage des fichiers et d'autres fonctionnalités (pour cela nous pensions utiliser Google Code), cependant, suite à quelques soucis dans le binôme (séparation), ces fonctionnalités n'étaient pas nécessaires. Un logiciel permettant la planification de projet a été utilisé, celui-ci nous a permis de bien organiser le projet, ainsi que de se fixer des objectifs à atteindre à des dates données. Le logiciel que nous avons utilisé est, lui aussi, un logiciel libre nommé Planner (*http ://www.simpleprojectmanagement.com/planner*)

2.5 Gestion du projet

Pour permettre le bon déroulement du projet, nous avons décidé en accord avec notre tuteur de se voir régulièrement pour pouvoir avoir un avis de la part de notre tuteur et donc être le plus proche possible de ce qu'il attendait. Pour cela, nous avons donc décidé de se voir toutes les semaines pendant à peu près une heure. Ces réunions étaient l'occasion de montrer le travail effectué, d'avoir un retour par rapport à cela, et de pouvoir organiser le travail à venir. De plus, cela permettait, d'une part de nous rassurer, mais également de rassurer notre tuteur qui pouvait grâce à cela être au courant de notre avancée.

De plus, un outil de planification a été utilisé, permettant l'organisation du travail en fonction des ressources disponibles et de pouvoir voir l'état du projet au cours du temps et augmenter la priorité de certaines tâches si besoin est (cf Fig. 3).

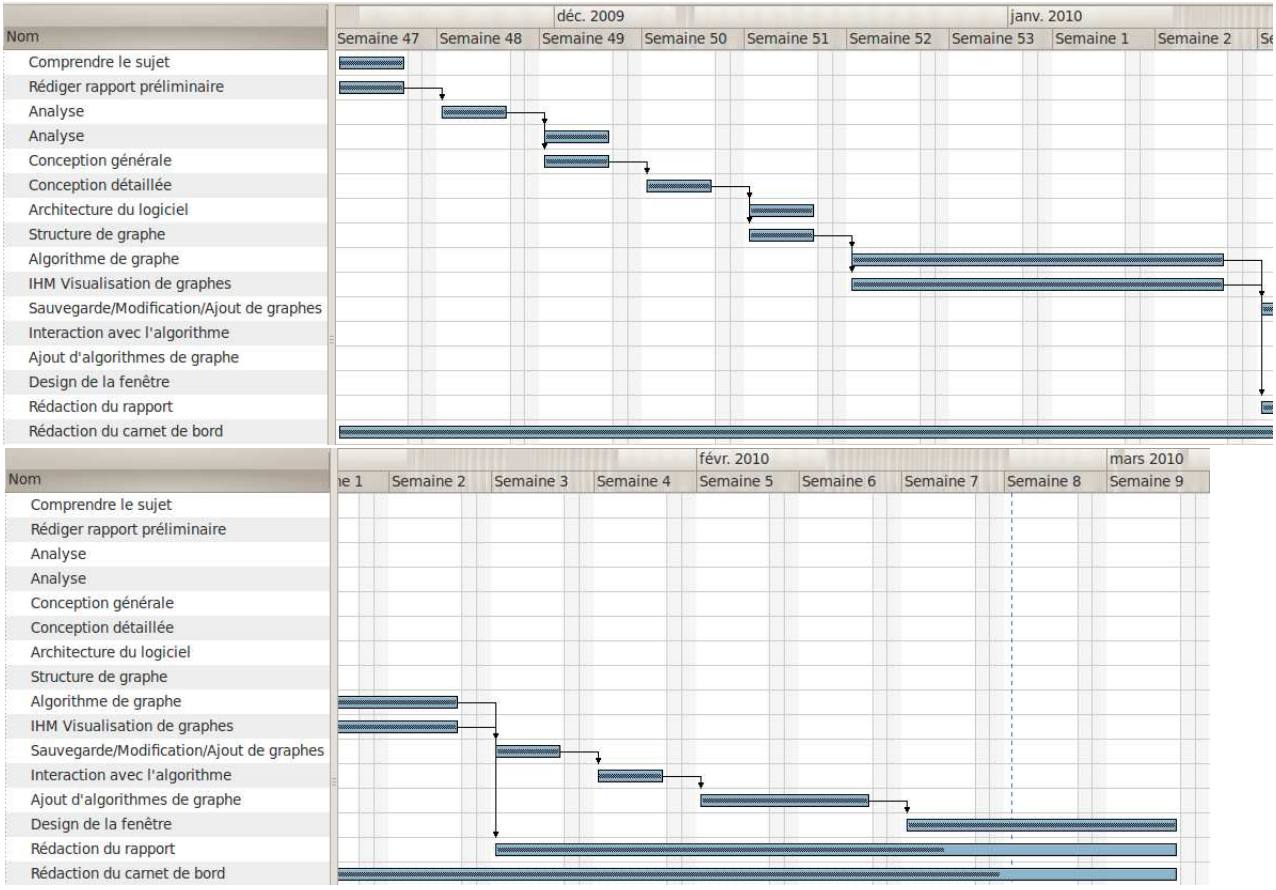

FIG. 3 – Planning réalisé pour le déroulement du projet

3 Conception

Afin de bien organiser le développement du logiciel nous avons dû séparer les différents composants du logiciel et établir une architecture basée sur le M.V.C. Pour ce faire, nous avons donc amélioré notre premier diagramme de classe pour créer l'architecture du logiciel.

3.1 Diagramme de classe

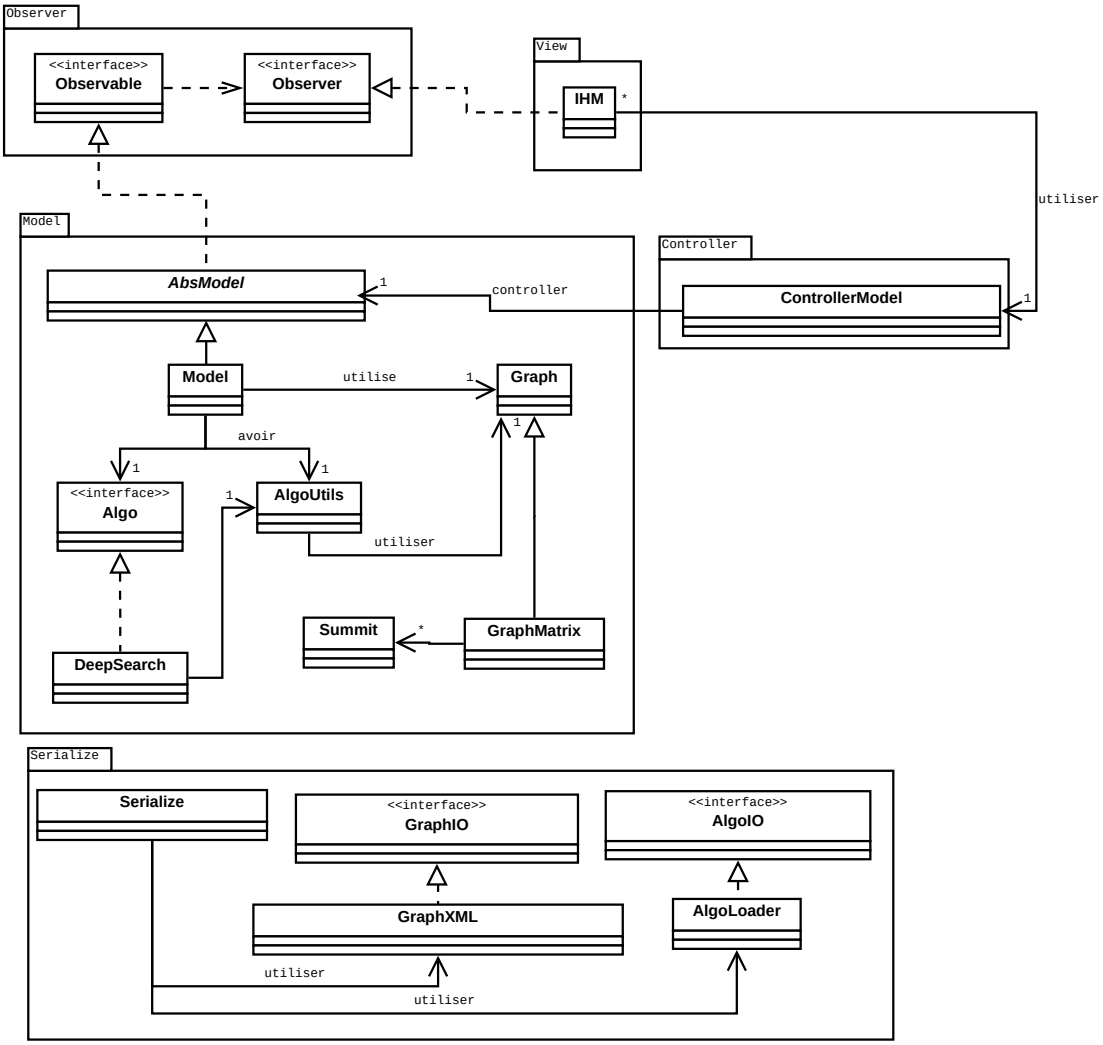

FIG. 4 – Diagramme de classe de conception (mode esquisse)

Diagramme de classe en mode plan fourni en annexe (Fig. 15, 16, 17, 18, 19)

3.2 Diagramme de package

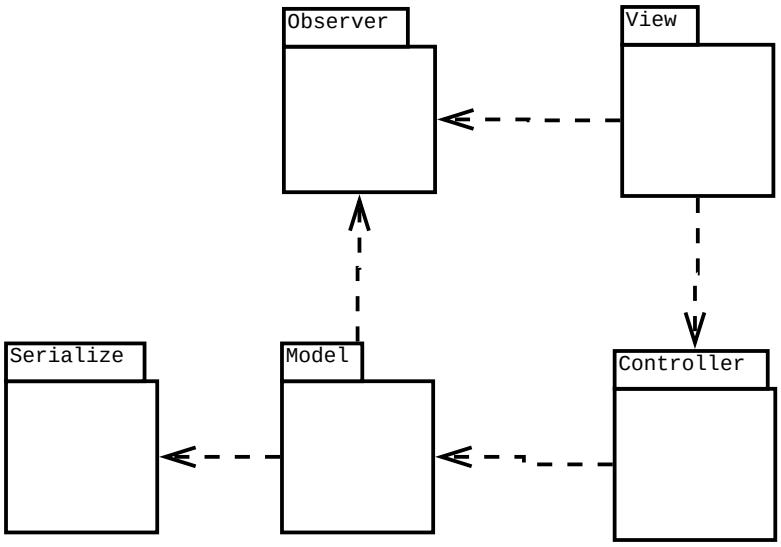

FIG. 5 – Diagramme de package

A l'aide de ce diagramme de package, on voit bien l'avantage du M.V.C, en effet, on se rend compte que la vue est totalement indépendante du modèle. Ceci peut donc s'avérer utile si on décide de changer d'interface graphique par exemple. Ceci est notamment utile si des ajouts ou des modifications sont effectuées sur le modèle. Il est donc possible d'ajouter des structures de graphes ou des algorithmes sans avoir à modifier quoi que ce soit sur notre interface graphique, et inversement, on peut modifier notre interface sans avoir à modifier notre modèle. Grâce à cela, on a donc un logiciel qui peut assez facilement évoluer sans avoir à retoucher tout le programme.

4 L'objet Graphe

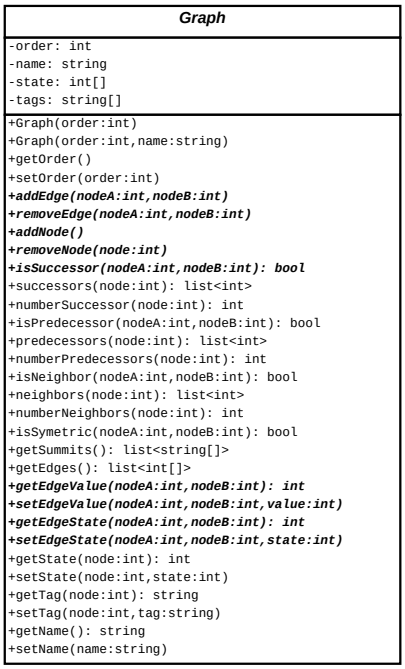

FIG. 6 – Classe Graph

4.1 Sauvegarde et importation

Dans le cahier des charges, il était stipulé que les graphes devraient pouvoir être sauvegardés. Pour ce faire, nous avons décidé d'effectuer la sauvegarde dans un fichier XML. En effet, la sérialisation XML est beaucoup utilisée que ce soit dans les logiciels (sauvegarde de données, propriétés liées au programme) ou bien les services web ou les technologies web. De plus, la sérialisation XML s'avère assez aisée à utiliser. Après quelques recherches concernant des parser XML en C++, nous avons découvert TinyXML (parseur XML libre et open source) qui s'avérait relativement simple à utiliser, s'appuyant sur la technologie DOM qui crée un arbre contenant l'ensemble du fichier XML. Grâce à cela, il est donc possible de naviguer dans l'arbre en parcourant les enfants d'un sommet. Nous avons donc défini une structure pour la sauvegarde d'un graphe qui est la suivante :

```
<graph name="">
    <order></order>
    <listNode>
        <node num="" tag="" state=""/>
        ...
    </listNode>
    <listEdge>
        <edge start="" end="" value="" state=""/>
        ...
    </listEdge>
</graph>
```

Grâce à cette structure, il nous est maintenant possible de sauvegarder un graphe, mais pas uniquement l'ensemble de ses nœuds et de ses arcs, mais également l'état de ceux-ci ainsi que leurs valeurs. A partir de cela, il est donc possible de restaurer un graphe avec l'état de chacun de ses nœuds et de ses arcs.

4.2 Structure de graphe utilisée

Le logiciel devait permettre l'ajout de nouvelles structures de graphe. A l'heure actuelle, seule la structure de graphe symbolisée par une matrice a été implémentée. D'autres structures sont possibles, en effet, un graphe peut se représenter en machine à l'aide de différentes structures comme :

- **Représentation par le biais d'une matrice** : Une matrice de contenant $n * n$ cases avec n étant l'ordre du graphe, et où chaque case i, j (avec $i, j \in 0..n$) contient une valeur booléenne permettant de savoir s'il y a oui ou non un arc entre le nœud i et le nœud j.
- **Représentation par le biais d'un tableau de liste de successeurs** : Dans cette représentation nous avons un tableau de taille n, avec n étant l'ordre du graphe, et chaque case i (avec $i \in 0..n$) contient l'ensemble des successeurs de i.
- **Représentation par une liste d'arcs** : Cette représentation est la moins utilisée des trois, elle contient uniquement un entier n représentant le nombre de nœuds du graphe, et une liste contenant les arcs du graphe.

Bien entendu, ces structures sont simplifiées et différentes implémentations sont possibles suivant les besoins de chacun.

Pour ce projet nous avons donc utilisé une structure représentée par une matrice, cependant des modifications ont été effectuées. En effet, notre matrice ne contient pas simplement une

valeur permettant de savoir si oui ou non il y a un arc, dans notre cas, chaque case i, j contient un entier négatif s'il n'y a pas d'arc entre i et j et un entier supérieur à 0 s'il y a un arc, et cette valeur correspond à la valeur de l'arc. De plus en se basant sur le même principe, nous avons une deuxième matrice qui nous permet de sauvegarder l'état des arcs et enfin nous avons une liste d'objets représentant un nœud, dans lesquels sont enregistrés l'état et l'étiquette de chaque nœud.

4.3 Ajout de structures de graphe

Grâce à la structure du logiciel, il est possible d'ajouter de nouvelles structures de graphe basées sur celles décrite précédemment ou éventuellement de nouvelles structures issu de réflexions diverses et variées. Pour ce faire, il suffit de faire hériter notre nouvelle structure de la classe Graph (cf Fig. 6). Cette classe contient différentes méthodes dont certaines abstraites qu'il faudra donc redéfinir dans notre nouvelle structure. Ces méthodes à redéfinir sont les suivantes :

- *void addEdge(int nodeA, int nodeB)* : Ajoute un arc reliant le nœud *nodeA* au nœud *nodeB* avec $nodeA, nodeB \in 0..n$.
- *void removeEdge(int nodeA, int nodeB)* : Supprime l'arc reliant *nodeA* à *nodeB* avec $nodeA, nodeB \in 0..n$.
- *void addNode()* : Ajoute un nœud au graphe.
- *void removeNode(int node)* : Supprime le nœud node du graphe avec $node \in 0..n$.
- *bool isSuccessor(int nodeA, int nodeB)* : Retourne vrai si le *nodeA* est un successeur de *nodeB*, faux sinon avec $nodeA, nodeB \in 0..n$.
- *int getEdgeValue(int nodeA, int nodeB)* : Retourne la valeur de l'arc $nodeA, nodeB$ avec $nodeA, nodeB \in 0..n$.
- *void setEdgeValue(int nodeA, int nodeB, int value)* : Assigne la valeur *value* à l'arc $nodeA, nodeB$ avec $nodeA, nodeB \in 0..n$.
- *int getEdgeState(int nodeA, int nodeB)* : Retourne l'état de l'arc $nodeA, nodeB$ avec $nodeA, nodeB \in 0..n$.
- *void setEdgeState(int nodeA, int nodeB, int state)* : Assigne l'état *state* à l'arc $nodeA, nodeB$ avec $nodeA, nodeB \in 0..n$ et $state \in 0..3$.

D'autres fonctions peuvent être redéfinies, cela n'est pas nécessaire, mais peut s'avérer utile lorsque de bonnes performances sont attendues ou simplement lorsque l'on en a envie. Une fois toutes les fonctions utiles redéfinies, la structure peut être utilisée avec n'importe quel algorithme.

5 L'objet Algorithme

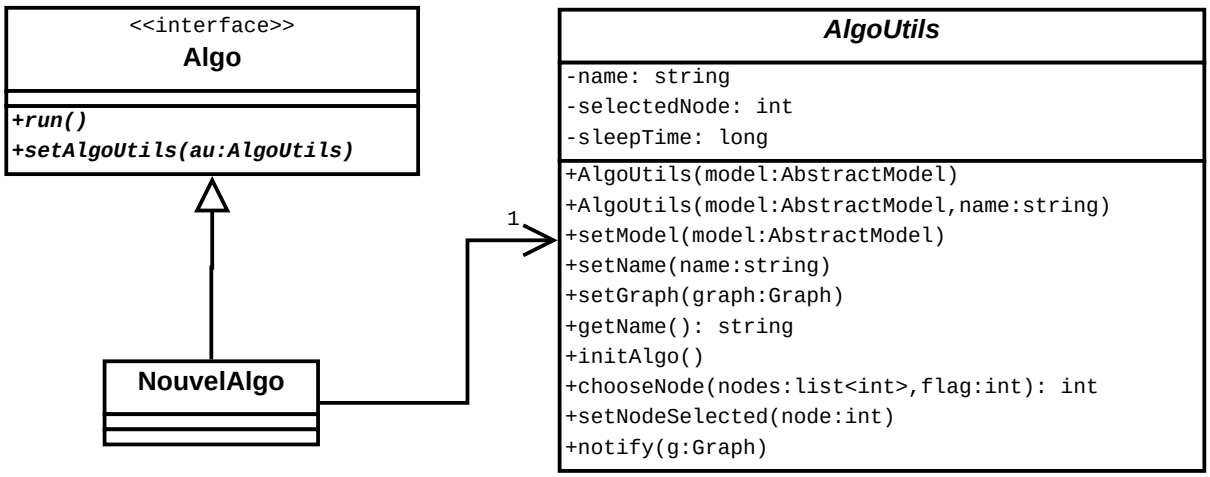

FIG. 7 – Classe Algo

5.1 Ajout d'algorithmes

Ce logiciel offre la possibilité d'y intégrer ses propres algorithmes. Afin que ceux-ci soient correctement utilisables, certaines contraintes sont à respecter. Dans un premier temps, comme on peut le voir sur le diagramme UML (Fig. 7), notre algorithme doit hériter de l'objet algo et doit de ce fait, redéfinir les fonctions *run(g :Graph)* et *setAlgoUtils(au :AlgoUtils*) qui sont respectivement les fonctions permettant de lancer l'algorithme sur le graphe g fourni en paramètre et la fonction permettant d'attribuer à notre algorithme quelques fonctions utiles, comme la fonction chooseNode() qui retourne un nœud choisi par l'utilisateur parmi une liste de nœuds fournie par l'algorithme. En plus de ces fonctions à implémenter et de l'héritage à faire, il est également nécessaire de rajouter un créateur et un destructeur externe à l'objet, ce qui sera utile lors du chargement de l'algorithme (cf Chargement dynamique des algorithmes). Ce créateur appelé *create* et retournant un objet Algo va avoir pour but de créer le nouvel algorithme (l'équivalent d'un new NouvelAlgo()). De même, pour le destructeur appelé *destroy* avec comme paramètre un objet Algo, permet de détruire l'algorithme lorsque nous n'en avons plus besoin.

Après tout cela, notre nouvel algorithme s'organise donc de la manière suivante :

NouvelAlgo.h	NouvelAlgo.cpp
♯ ifndef NOUVELALGO_H	♯ include "NouvelAlgo.h"
♯ define NOUVELALGO_H	NouvelAlgo : :NouvelAlgo(){}
♯ include "Algo.h"	NouvelAlgo : : NouvelAlgo() {}
class NouvelAlgo : public Algo {	void NouvelAlgo : :run(Graph *g) {}
public :	void NouvelAlgo : :setAlgoUtils(AlgoUtils *au) {
NouvelAlgo() ;	this→au = au ;
virtual NouvelAlgo() ;	}
run(Graph ∗g) ;	extern "C" Algo* create(){
setAlgoUtils(AlgoUtils ∗au) ;	return new NouvelAlgo ;
private :	}
AlgoUtils ∗au ;	extern "C" void destroy(Algo* a) {
} ;	delete a ;
♯endif	}

Libre à vous par la suite de créer de nouvelles fonctions ou attributs à l'intérieur de l'algorithme ne servant qu'à celui-ci.

Ces fichiers doivent être placés dans le répertoire model/Algorithm du projet afin que les dépendances soient satisfaites.

5.2 Chargement dynamique des algorithmes

L'une des principales contraintes de ce logiciel était de le faire évolutif. Pour cela, nous avons permis l'ajout d'algorithme (voir chapitre précédent). Pour ne pas avoir à recompiler à chaque fois l'ensemble du logiciel pour l'ajout d'un algorithme nous voulions faire ceci à la manière de greffon. En effet, chaque algorithme sera un greffon de notre logiciel, et nous pourrons donc ajouter de nouveaux greffons facilement sans avoir à effectuer des modifications sur le logiciel. Tout se fera automatiquement et nous aurons seulement à ajouter le fichier source de notre algorithme ou bien la librairie (.so sous linux et .dll sous windows).

Cette partie a été la plus difficile à réaliser car si cela est plutôt simple à réaliser en Java (deux lignes suffisent), cela se complexifie en C++. Après de multiples recherches nous avons remarqué que l'héritage était très mal géré, c'est pour cela que nous avons dû dissocier l'objet Algo et l'objet AlgoUtils qui étaient, dans un premier temps, un seul et même objet. Grâce à cette modification, nous avons pu charger nos algorithmes grâce à la librairie *dlfcn.h* sous système Unix et windows.h sous système Windows. Ces librairies permettent de charger une librairie partagée grâce à la fonction *dlopen()* (*LoadLibrary()* sous Windows). Une fois cette librairie chargée, les fonctions

de l'algorithme sont récupérées grâce à la fonction *dlsym()* (*GetProcAddress()* sous Windows) qui récupère un pointeur sur une fonction déclarée externe à l'objet. C'est pour cela que lors de la création d'un nouvel algorithme, il est nécessaire d'avoir nos fonctions *create()* et *destroy()* qui sont externes, cela permet de créer l'objet afin de l'utiliser par la suite, et pouvoir le détruire après. Et enfin il ne faut pas oublier de fermer la librairie que nous venons d'ouvrir à l'aide de la fonction *dlclose()* (*FreeLibrary()* sous Windows). Grâce à ces fonctions, nous pouvons charger nos nouveaux algorithmes sans que l'ajout de ceux-ci nécessite une recompilation du logiciel tout entier. Nous pouvons donc maintenant simplement ajouter un algorithme par le biais de l'interface graphique très facilement. Il suffit pour cela de créer notre source de l'algorithme en suivant les contraintes vues dans le paragraphe précédent et de l'ajouter, par le biais de l'interface graphique qui fera le travail de compilation en librairie dynamique et le chargera dans l'interface graphique. Celui-ci sera donc utilisable sans même avoir à redémarrer le logiciel.

5.3 Algorithmes implémentés

Afin de fournir une version que l'on puisse utiliser, le logiciel devait être fourni avec une bibliothèque d'algorithmes. Pour ce faire, nous avons donc développé deux algorithmes. Ces algorithmes ont été choisis car ce sont des algorithmes enseignés lors de l'apprentissage des graphes. En effet, cette année, pour nous familiariser avec les graphes, nous avons analysé ces algorithmes. Ces algorithmes ne présentent pas de grosses difficultés et il est relativement simple de pouvoir faire la trace de ceux-ci en utilisant des petits exemples.

5.3.1 Descente en profondeur

Premièrement, nous avons développé une descente en profondeur. Cet algorithme permet de connaître tous les descendants d'un nœud, c'est-à-dire, de manière plus vulgarisé, l'ensemble des points vers lesquels on peut aller en suivant les flèches. Ce type d'algorithme peut être utile dans de multiples cas. Si on prend comme exemple un réseau de communication, la descente du graphe permettra de connaître les points vers lesquels on peut communiquer.

Cet algorithme de descente peut être implémenté en suivant diverses méthodes et de diverses façons. Tout d'abord, deux méthodes s'offrent à nous :

- Depth-first search (Recherche en profondeur) : Cet algorithme va parcourir le graphe en allant le plus loin possible et en remontant pour explorer les nouvelles branches pas encore parcourues.
- Breadth-First Search (Recherche en largeur) : Ici, l'algorithme va lister les nœuds vers lesquels il peut aller pour ensuite les parcourir un par un.

Pour implémenter cet algorithme, nous avons décidé de faire une descente en profondeur dont vous pouvez voir l'algorithme ci dessous.

Descente_en_profondeur(entier n)

Données : Un nœud n appartenant au graphe g

Résultat : Un ensemble contenant les descendants de n appartenant au graphe g

début

```
pour chaque successeurs succ de n dans g faire
    si succ non marqué alors
        sauvegarde du descendant succ
        marquer succ dans g
        Descente_en_profondeur(succ)
```

fin

Algorithme 1 : Algorithme de descente en profondeur

5.3.2 Tri topologique

Le deuxième algorithme implémenté est celui du tri topologique, également appelé numérotation compatible. Cet algorithme permet de parcourir le graphe de manière à ce qu'un sommet soit toujours parcouru avant ses successeurs. Afin de parcourir l'ensemble des nœuds du graphe, il est nécessaire que celui-ci soit acyclique, c'est-à-dire qui ne contienne pas de cycle et donc qu'on ne puisse pas, à partir d'un nœud du graphe, revenir à ce nœud. Cependant, si nous avons un tel graphe G contenant un ou plusieurs cycles, l'algorithme fonctionnera mais parcourra uniquement les nœuds du sous graphe acyclique de G.

Un tel algorithme peut avoir différentes utilités. Par exemple, nous avons une liste de tâche à effectuer et certaines tâches doivent être effectuées dans la continuité d'autres tâches pour leur bon déroulement. Si nous utilisons un graphe où un nœud représente une tâche et un arc représente la dépendance entre deux tâches, cet algorithme permettra de savoir par quelle(s) tâche(s) nous pouvons commencer et l'enchaînement des tâches suivantes.

Vous pouvez voir cet algorithme ci dessous.

Tri_topologique()

Données : Un graphe g

Résultat : La numérotation compatible de g

début

- $i = 0$
- copier le graphe g dans g'
- **tant que** *il reste des sources dans* g' **faire**
  - choisir une source src de g'
  - val(src dans g) $= i$
  - supprimer src de g'
  - $i++$

fin

Algorithme 2 : Algorithme de tri topologique

6 Interface graphique

6.1 Représentation d'un graphe

Afin de pouvoir bien visualiser un graphe, nous ne pouvions pas nous contenter de simplement afficher les nœuds sans leur affecter de positions précises permettant de ne pas surcharger la représentation. Dans un premier temps, nous avons donc pensé à disposer les nœuds sur un cercle où chaque nœud serait espacé l'un de l'autre, permettant une visualisation assez claire d'un graphe (Fig. 8). Bien que cette représentation s'avérait assez compréhensible, cela ne nous convenait pas totalement. Nous avons donc effectué quelques recherche et avons décider d'utiliser un des exemple de la librairie Qt4 appelée Elastic Node (*http ://doc.trolltech.com/4.3/graphicsview-elasticnodes.html*), qui permet de calculer des forces entre les nœuds, ce qui a pour conséquence de les repousser entre eux et donc de les espacer au maximum, permettant de visualiser le graphe plus facilement et d'en extraire les sous-graphes non connexes, qui lors du calcul des forces, va donc repousser tous les nœuds du sous-graphe qui ne sont pas reliés au graphe par au moins un arc. Cette méthode a donc été couplée à la première, c'est-à-dire que les nœuds ont donc été disposés sur un cercle et par la suite, les forces entre les nœuds ont été calculés. Cela nous permet donc d'avoir une représentation du graphe assez lisible (Fig. 9).

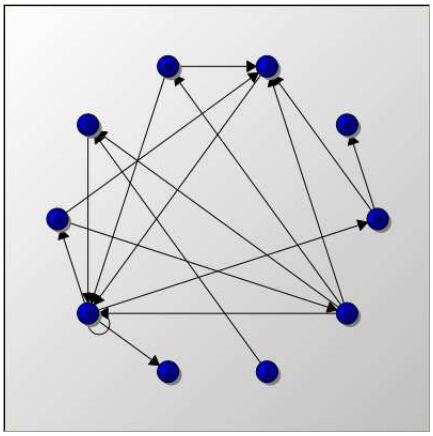

FIG. 8 – Représentation d'un graphe disposé sur un cercle

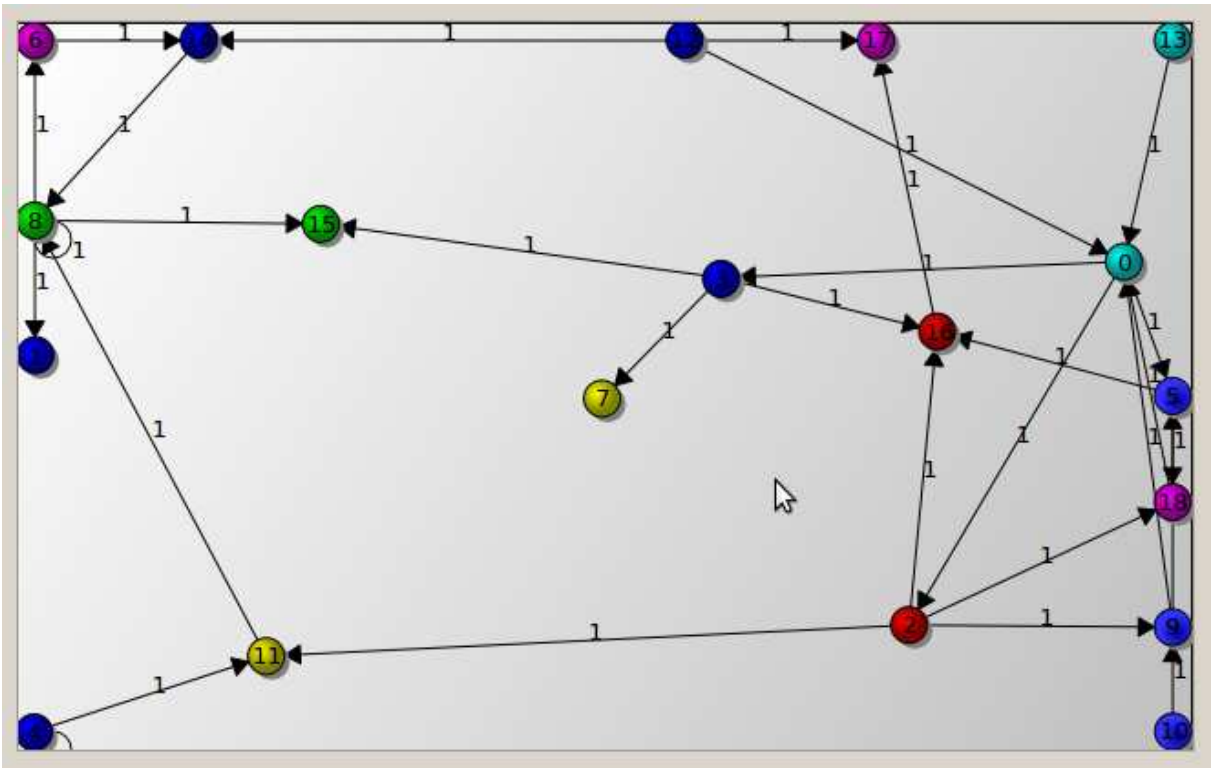

FIG. 9 – Représentation d'un graphe

6.2 Fonctionnalités proposées

Le logiciel propose différentes fonctionnalités. Ces fonctionnalités ont donc du être facilement accessibles au niveau de l'interface graphique. Pour ce faire, nous avons utilisé toutes sortes de composants graphiques telles que des barres d'outils, des barres de menus ou des boites à outils permettant d'accéder aux fonctionnalités suivantes :

- **Création d'un graphe** : Grâce à cette fonctionnalité il est possible de créer ses propres graphes à l'aide de l'interface graphique. Pour cela, il suffit de double cliquer pour ajouter un nœud. Pour ajouter des arcs, il suffit de cliquer une première fois sur la source de l'arc, et une deuxième fois sur la destination de l'arc.

  Accessible via le menu (Fig. 10) Fichier → Nouveau graphe ou bien via la première icône de la barre d'outils (Fig. 11).
- **Génération de graphes aléatoires** : Un graphe peut être généré de façon aléatoire. Ce graphe sera constitué d'un ensemble de nœuds allant de 0 à n avec $n <= 20$ et d'un ensemble d'arcs compris entre 0 et n^2.

  Accessible via le menu (Fig. 10) Fichier → Générer un graphe ou bien via la deuxième icône de la barre d'outils (Fig. 11).
- **Exportation et importation de graphe** à partir d'une librairie de graphe ou bien d'un

fichier : Le logiciel permet la sauvegarde et l'ouverture de graphes. Grâce à cela, il est donc possible de charger des graphes présents dans une librairie de graphes fournis avec le logiciel, ou bien de charger un graphe à partir d'un fichier. Il est également possible de sauvegarder nos graphes préférés en mémorisant l'état de chacun de ses sommets et de ses arcs (si un algorithme est en cours d'exécution sur ce graphe, l'état de l'algorithme ne sera pas sauvegardé). Accessible via le menu Fichier (Fig. 10) ou bien via la deuxième partie de la barre d'outils (Fig. 11).

- **Ajout et modification de nœuds et d'arcs** : Comme pour la création d'un graphe, il est possible de modifier un graphe en lui ajoutant de nouveaux nœuds ou de nouveaux arcs ou bien en modifiant ceux-ci sur n'importe quel type de graphe (provenant de la libraire de graphes, d'un graphe importé ou bien d'un graphe créé par l'utilisateur).
  Accessible via la scène graphique (Fig. 12).
- **Type d'affichage du graphe** : Cette fonctionnalité permet de régler différents paramètres concernant la représentation du graphe comme l'affichage des valeurs des arcs ou bien l'animation du graphe, ce qui permet de faire bouger le graphe pour le rendre le plus clair possible.
  Accessible via la boite d'outils (Fig. 13).
- **Sélection d'algorithmes** : Le logiciel permet d'exécuter différents algorithmes en rapport avec les graphes. Il est donc possible de choisir un algorithme parmi la liste des algorithmes fournis par le logiciel et de l'exécuter.
  Accessible via la dernière partie de la barre d'outils (Fig. 11).
- **Lancement d'algorithmes** : Permet de lancer l'algorithme sélectionné dans la barre d'outils (Fig. 11) et permet de visualiser son déroulement sur la scène graphique (Fig. 12).
  Accessible via la boite d'outils (Fig. 13) ou bien par le menu Algorithme (Fig. 10).
- **Type d'interaction avec l'algorithme** : Il est possible d'interagir avec un algorithme lors de son déroulement. Ce type d'interaction peut varier. L'utilisateur peut cliquer sur les nœuds qu'il veut ou bien laisser l'ordinateur choisir. Les choix de l'ordinateur sont de différents types :
  - Fifo, qui permet de sélectionner le premier nœud de la liste des nœuds disponible
  - Lifo, qui choisit le dernier nœud
  - Aléatoire, qui choisit un nœud de façon aléatoire parmi les nœuds disponibles.

  Accessible via la boite d'outils (Fig. 13).
- **Vitesse de défilement de l'algorithme** : Lorsque le type d'interaction permet à l'ordinateur de choisir ses propres nœuds. Il est nécessaire de pouvoir voir les choix de l'algo-

rithme. Pour ce faire, on peut indiquer à l'algorithme le temps qu'il doit attendre avant de prendre une nouvelle décision.
Accessible via la boite d'outils (Fig. 13).

FIG. 10 – Barre de menu

FIG. 11 – Barre d'outils

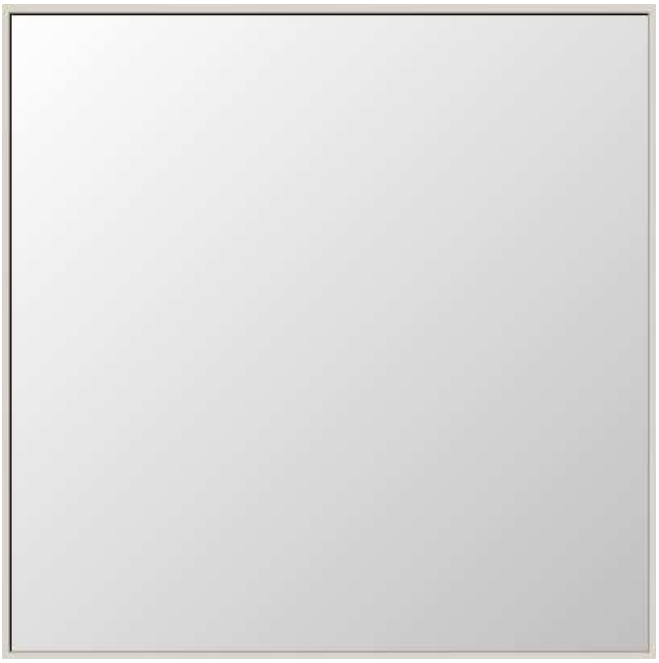

FIG. 12 – Scène graphique

FIG. 13 – Boite à outils

7 Interaction entre l'interface graphique et l'algorithme

Pour donner un peu plus de dynamisme au logiciel, l'utilisateur devait pouvoir interagir avec l'algorithme qu'il avait lancé. Cette fonctionnalité permet d'indiquer à l'algorithme les nœuds par lesquels il doit passer, le tout en ayant un aperçu, au cours de l'algorithme, des choix que l'on a indiqué à l'algorithme et des opérations que celui-ci a effectuées. Pour cela, nous avons donc d'abord utilisé le pattern de conception observateur, qui permet de mettre à jour, dans notre cas, des composants graphiques dès qu'on en a besoin et ensuite temps des threads permettant l'utilisation de l'algorithme et de l'interface graphique en parallèle.

7.1 Le pattern Observateur

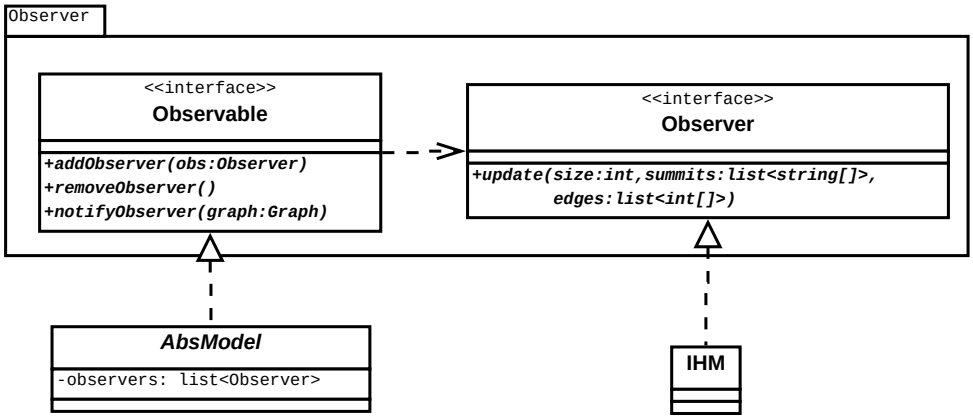

Fig. 14 – Pattern observateur implémenté

Comme on peut le voir à l'aide du diagramme UML (Fig. 14), dans cette implémentation du pattern observateur que nous avons utilisé, lorsque nous modifions notre graphe, cette modification sera suivie de l'appel de la fonction *notifyObserver(graph :Graph)*, dans laquelle la paramètre fourni sera le graphe avec ses nouvelles modifications. Cette fonction va donc parcourir l'ensemble des *observers* (qui sont en réalité des composants de notre fenêtre, voire la fenêtre elle-même) et leur envoyer le nouvel état du graphe. Une fois ces informations reçues, chaque *observer* va traiter ces informations en fonction de ce dont elle a besoin. Pour ce projet, un seul *observer* existe pour l'instant, qui se trouve être la représentation graphique du graphe (Fig. 12). A l'aide de ce procédé, l'utilisateur peut maintenant effectuer des modifications sur le graphe et avoir une représentation de cette modification.

En quoi ceci permet t-il d'interagir avec l'algorithme ?

Lors du déroulement de l'algorithme, celui-ci va proposer une liste de nœuds que l'utilisateur pourra choisir. Pour ce faire, l'algorithme va affecter à ces nœuds un état prédéfini pour la

sélection. Lorsque l'interface graphique rencontre un nœud avec cet état là, elle permet le clic sur ce nœud, ce qui prévient l'algorithme que le nœud en question a été choisi et l'algorithme peut donc continuer.

7.2 Utilisation de Threads

Lorsque l'algorithme propose une liste de nœuds à l'utilisateur, celui-ci doit attendre le choix de l'utilisateur. Ceci crée un problème. En effet, étant donné que l'algorithme a été lancé par le biais de l'interface graphique et que cet algorithme est maintenant en attente, cela implique donc que l'interface graphique est également en attente et ne peut donc pas gérer les évènements de clics, par exemple pour choisir un nœud. Il est donc impossible de pouvoir choisir un nœud que propose l'algorithme, car tant que l'algorithme n'est pas terminé, l'interface graphique reste figée. Pour régler ce problème, il serait donc intéressant de pouvoir lancer l'algorithme en parallèle de l'interface graphique et que les deux processus puissent s'exécuter séparément, mais tout en gardant une communication entre eux. Nous aurions pu lancer l'algorithme dans un nouveau programme, mais nous aurions perdu la communication entre l'interface graphique et l'algorithme. Ceci n'était donc pas une bonne solution.

Après avoir bien cerné le problème, nous avons donc eu l'idée de threader notre programme. En effet, si nous lançons un algorithme dans un nouveau thread, cela aurait pour conséquence d'exécuter en parallèle notre algorithme et notre interface graphique et donc, lorsque l'algorithme attendrait que l'utilisateur choisisse un nœud, il n'empêcherait pas la fenêtre de se mettre à jour et de gérer ses évènements. L'utilisateur aura donc la possibilité d'effectuer diverses opérations sur la fenêtre, comme par exemple, choisir un nœud.

A cet instant le logiciel était en mesure de lancer un algorithme, de mettre en attente celui-ci tout en ayant une interface graphique toujours fonctionnelle. Cependant, comment faire pour que l'algorithme sorte de son état d'attente ? Pour cela, nous avons donc eu l'idée d'affecter un état à l'algorithme en fonction du choix de l'utilisateur. Il existe deux types d'états :

- En attente lorsque l'utilisateur n'a pas encore choisi de nœud.
- Actif lorsque l'utilisateur a choisi un nœud et cet état contient également la valeur du nœud sélectionné.

A l'aide de cela, il a été facile de gérer l'attente de l'algorithme car celui-ci entre dans une boucle qui le fait attendre tant que l'état de l'algorithme n'est pas actif, et le changement d'état se fait lorsque l'utilisateur clique sur un nœud. Cette technique se rapproche de l'utilisation de sémaphores permettant de gérer l'état d'un thread, mais avec, ici, en plus, la valeur désirée pour que l'algorithme puisse continuer.

Grâce à cela notre utilisateur est donc en mesure de lancer un algorithme, puis d'indiquer à l'algorithme les nœuds par lesquels il doit passer.

Grâce à l'utilisation de ce thread, il est donc également possible de modifier en plus de l'état du graphe, les caractéristiques de l'algorithme, comme par exemple, le type d'interaction ou bien la vitesse de défilement de celui-ci (cf Interface Graphique > Fonctionnalités proposée) pendant son déroulement.

8 Discussions

Maintenant que ce rapport arrive à sa fin, prendre un peu de recule sur le projet est essentiel. Concernant les objectifs du cahier des charges, ceux-ci ont été remplis, certains avec un peu plus de difficultés que d'autres, mais cela n'a pu être que bénéfique. Grâce à ces quelques difficultés, nous avons pu apprendre à maitriser de nouvelles techniques et outils. Cela nous a donc été grandement profitable. Nous avons réussi à faire un logiciel assez évolutif, dans lequel il est possible de rajouter des structures de graphe et des algorithmes, facilement.

Le déroulement du projet s'est assez bien passé, le planning prévisionnel était bien fait. En effet, celui-ci était en adéquation avec le déroulement du projet.

Concernant d'éventuelles améliorations, plusieurs peuvent être envisagées. Dans un premier temps, développer plus d'algorithmes afin d'avoir une vraie librairie d'algorithmes contenant une dizaine d'algorithmes différents. Nous pourrions également améliorer l'interface graphique qui est assez simpliste, mais fonctionnelle. La grosse amélioration possible concerne la configuration de l'algorithme comme le type d'interaction avec l'algorithme ou bien la vitesse de celui-ci. En effet, le type d'interaction propose différents choix (Utilisateur, Aléatoire, Fifo, Lifo) seulement, suivant l'algorithme développé, celui-ci ne propose pas forcément un choix donc cet outil peut être inutile. Il serait donc envisageable d'afficher ce composant uniquement en fonction de l'algorithme choisi. De même, pour le choix de la vitesse, cette vitesse est exprimée en secondes. Suivant l'utilisation qu'on fait de ce logiciel, il peut être intéressant que la vitesse soit exprimée dans une, voire plusieurs autres unités de temps. Enfin, il serai bon de rajouter la sauvegarde de la configuration du logiciel afin que l'utilisateur puisse retrouver les choix qu'il a fait lors de la dernière utilisation du logiciel.

Concernant la création du graphe, celle-ci se fait uniquement à l'aide de clics au sein de l'interface graphique avec un double clic pour créer un nœud et deux clics successifs entre deux nœuds pour créer un arc. Cette méthode peut déplaire à certains utilisateurs.

Tout cela n'était pas précisé dans le cahier des charges, de ce fait, seul un test auprès de différents utilisateurs du logiciel pourrait permettre de les définir de façon plus précise.

9 Conclusion

Pour conclure, ce projet a été très intéressant. Tout d'abord car il nous a permis de mettre en pratique certaines connaissances découvertes lors de nos cours et plus particulièrement concernant les design pattern tel que la programmation en couches et l'utilisation du pattern M.V.C. ainsi que des fabriques abstraites. Ce projet nous a également permis de pouvoir planifier nos tâches et de voir que cela n'était pas forcément aisé de diriger un projet. Concernant l'équipe, cela s'est, hélas, assez mal passé. En effet, suite à très peu d'investissement dans le projet de la part de mon binôme, nous avons été contraint de nous séparer. Ceci à donc créer quelques problèmes, le premier étant bien sûr la charge de travail. En effet, le planning était fait pour deux personnes, après cette séparation je me suis retrouvé seul pour faire le projet.

Dans l'ensemble ce projet a apporté beaucoup, notamment l'apprentissage du chargement dynamique d'objets en C++, qui n'est pas une mince affaire, la création d'une fabrique permettant notamment la sauvegarde et l'importation de graphes en XML. De plus, le développement s'est fait en C++, ce qui nous a permis d'apprendre à maitriser un peu plus ce langage, ainsi que la librairie graphique Qt qui est une librairie beaucoup utilisée.

Au final, ce projet a été très bénéfique, que ce soit d'un point de vue programmation, organisation ou même réflexion. Il nous a permis de faire diverses découvertes, le sujet était plaisant et le fait de savoir que ce logiciel pourrait être utilisé par la suite a été très motivant.

10 Annexes

10.1 Diagramme de classe

Modèle

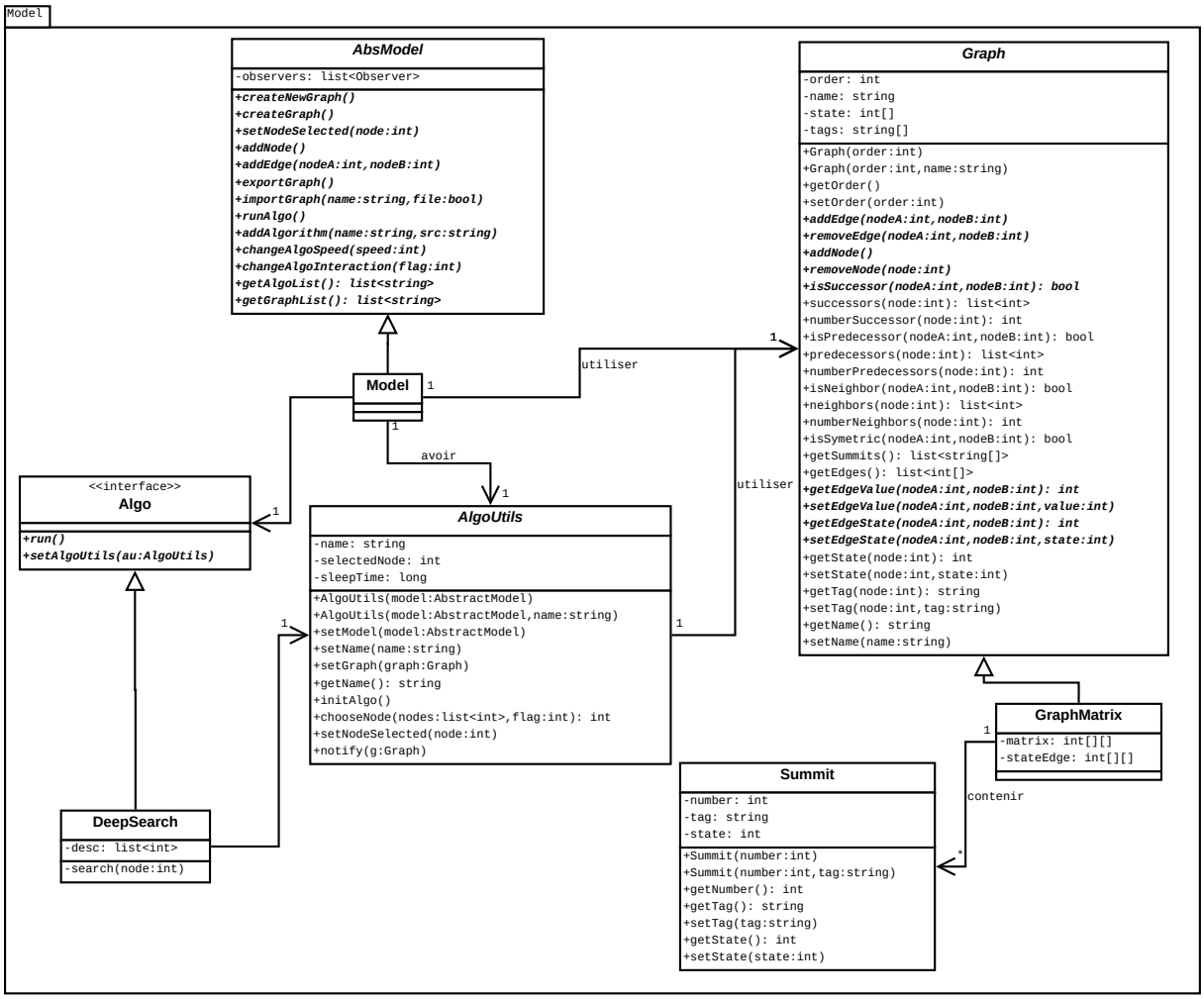

FIG. 15 – Package modèle

Sérialisation

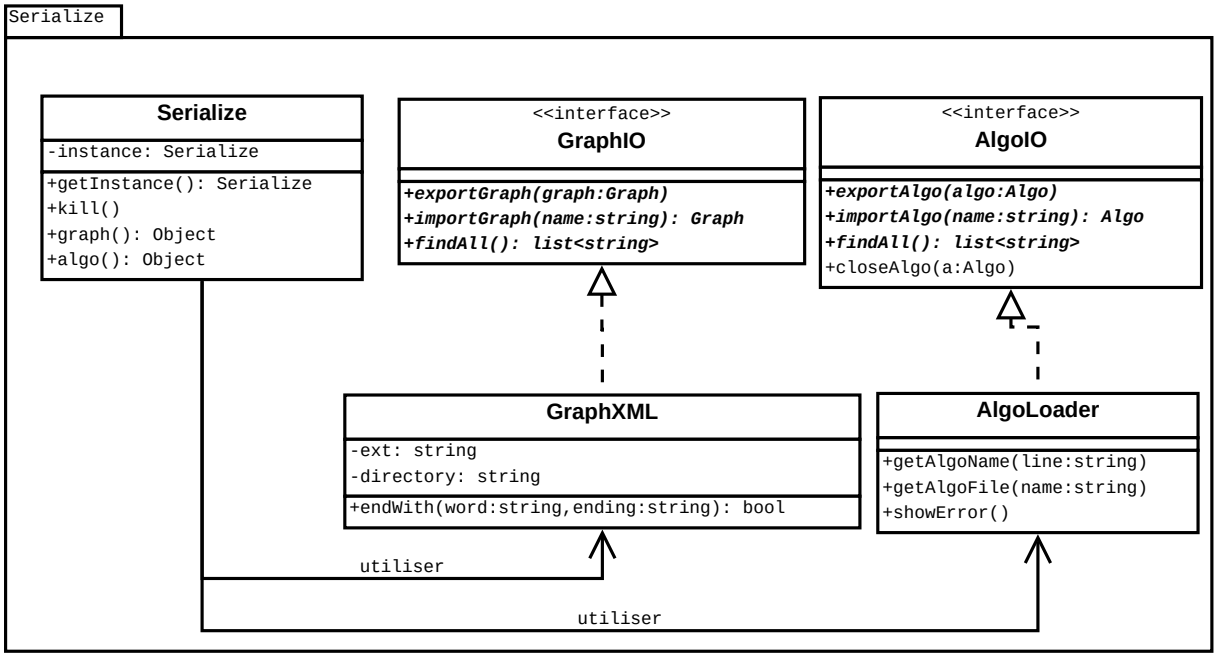

FIG. 16 – Package sérialisation

Contrôleur

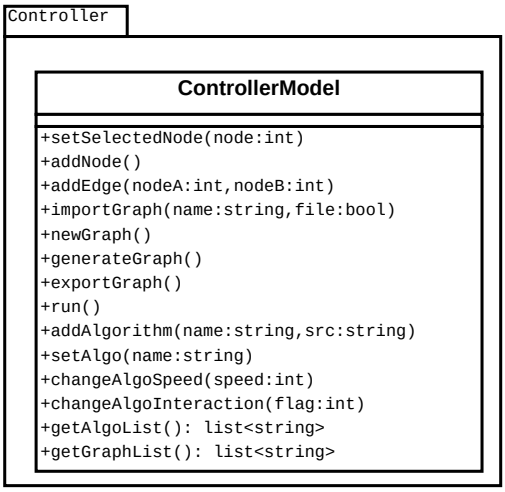

FIG. 17 – Package contrôleur

Observateur

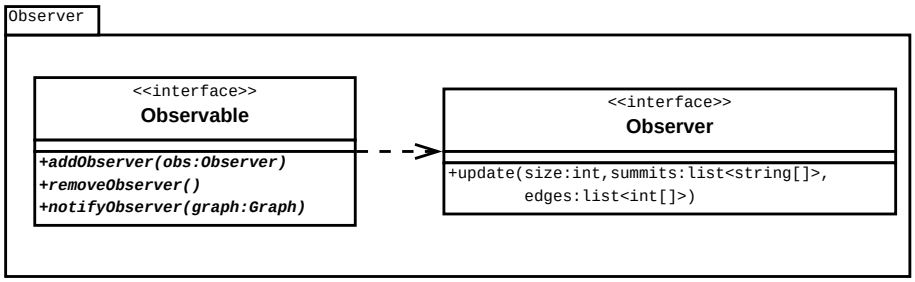

FIG. 18 – Package observateur

Interface Graphique

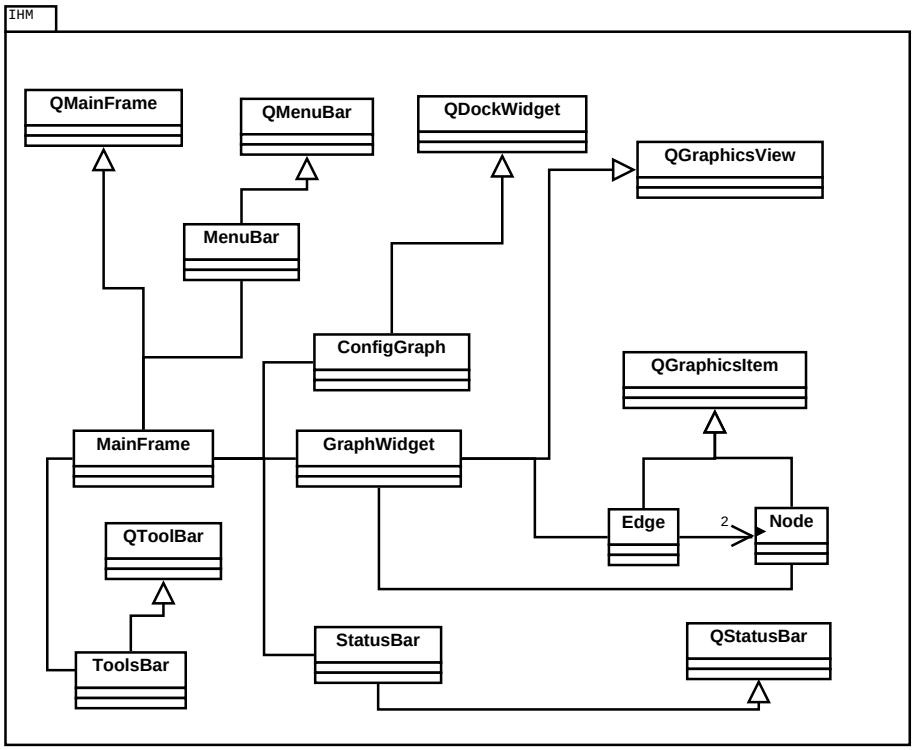

FIG. 19 – Package interface graphique

10.2 Bibliographie et sitographie

De manière générale :

- *www.google.fr*
- *fr.wikipedia.org/*
- *www.siteduzero.com/*

Page relative à la librairie Qt :

- *doc.trolltech.com/4.6/index.html*

Pages relatives au chargement dynamique des algorithmes :

- *www.linux-kheops.com/doc/man/manfr/man-html-0.9/man3/dlopen.3.html*
- *hiko-seijuro.developpez.com/articles/bibliotheque-dynamique/*
- *en.wikipedia.org/wiki/Dynamic-link_library*
- *en.wikipedia.org/wiki/Dynamic_loading*
- *www.faqs.org/docs/Linux-mini/C++-dlopen.html*

Pages relatives à la sérialisation xml :

- *www.grinninglizard.com/tinyxmldocs/index.html*

Resumé

Ce projet sous la tutelle de Mr Cogis a pour objectif de créer un logiciel pédagogique pour la compréhension d'algorithmes de graphes. Celui-ci, permettra à un utilisateur de créer ou visualiser des graphes et d'appliquer des algorithmes sur ceux-ci. Au cours de l'un de ces algorithmes, l'utilisateur aura la possibilité de choisir ses nœuds et de ce fait, pourra guider l'algorithme. Ces algorithmes pourront être ajoutés à la manière de greffons. Ainsi, les utilisateurs pourront développer leurs propres algorithmes et les ajouter facilement.

Ce logiciel pourra donc servir dans divers domaines et notamment dans l'enseignement lors de l'apprentissage des graphes ou bien d'algorithmes de graphes.

Abstract

This project under the direction of Mr Cogis aims to create an educational software to understanding graphs algorithms. This allows a user to create or view graphs and perform algorithms on them. During these algorithms, users have the opportunity to choose nodes and therefore can direct the algorithm. These algorithms can be added like plugins. The users can develop their own algorithms and add them easily.

This software can be used in several domains, particularly in education to learn graph or graph algorithms.