VANET Sullivan

HAUSER Guillaume

LADENBURGER Robin

**Rapport Projet R.O Modélisation**

**L3 Informatique**

**I Structure du graphe.**

La structure qui va nous permettre de d'instancier des graphes a été fait de la manière suivante:

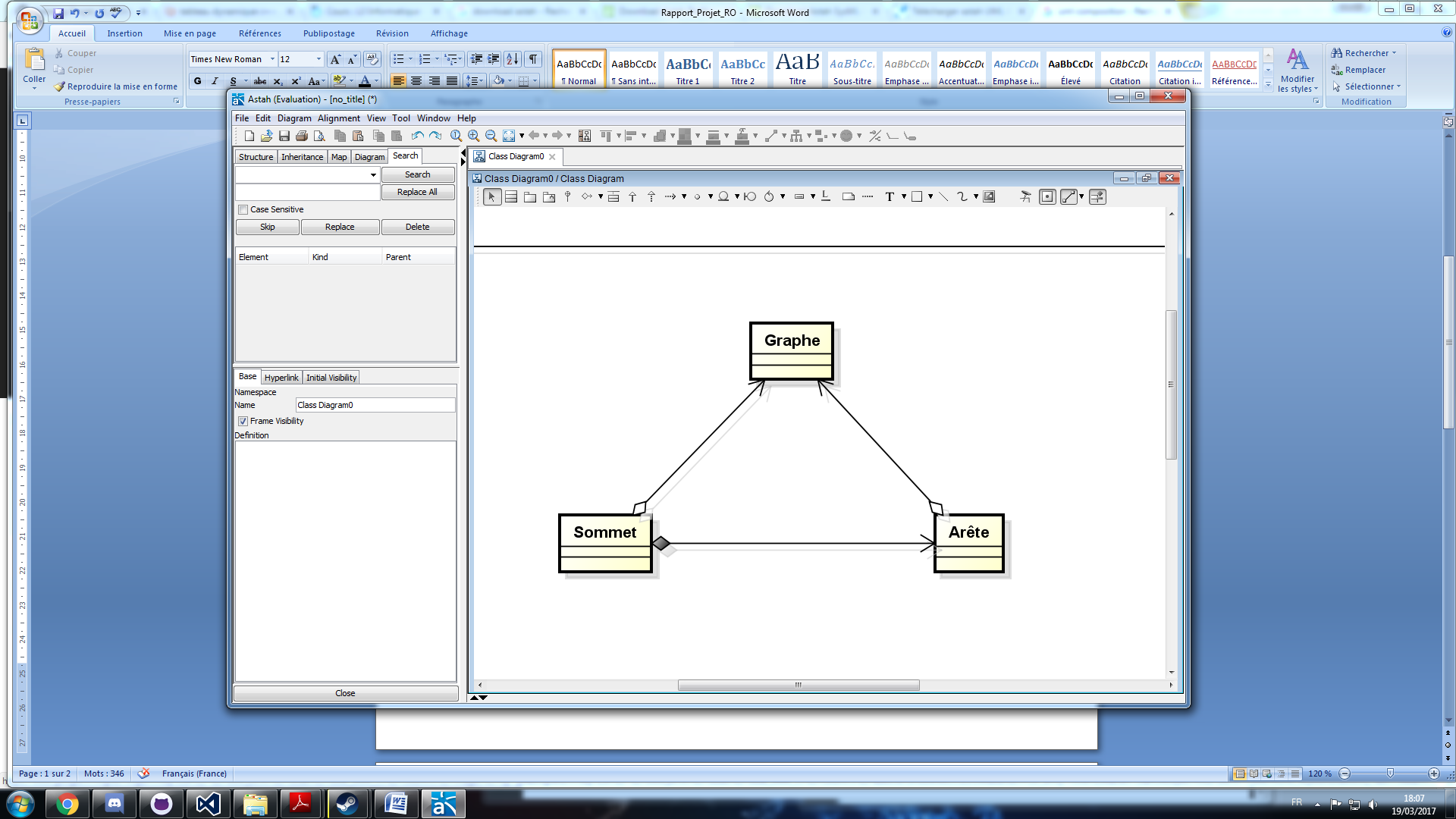
* On a une classe Sommet qui nous servira à instancier tout les sommets d'un graphe. Cette classe possède plusieurs attributs comme : son nombre de successeurs / prédécesseurs, son marquage, son nom, son poids, etc.
* On a une classe Arête nous servira à instancier tout les arêtes d'un graphe.

Elle possède deux sommets (un sommet "début" et un sommet "fin"),

Elle possède aussi un attribut cout (pour la valeur de l'arête) et un temps.

* Et enfin il y a une classe Graphe qui est décomposé en deux collections de sommets et d'arêtes. C'est dans cette classe que seront implémentés toutes les algorithmes que l'on a vus en cours ainsi que les méthodes nécessaires au fonctionnement des algorithmes.

Voici le diagramme UML de notre structure :



**II Génération des instances de graphes.**

Il a fallu gérer plein de problèmes car les GPR n'avaient pas de format standard.

Premièrement on crée un graphe vide, puis on lit le fichier ligne par ligne grace à la fonction "getline()" pour chaque ligne on parcourt tant qu'on ne trouve pas celles recherchées , on lit tous les sommets jusqu'à tomber sur une ligne "sources" et on crée tous ces sommets avec les informations du GPR que l'on récupère en lisant la ligne mot par mot grâce à la fonction "strtok" où l'on a mis comme séparateur l'espace puis on continue à parcourir jusqu'à trouver les lignes "puits", "sectionArcs", "sectiongraphes" et la fin de fichier.

La création des arêtes se fait de la même manière que les sommets, les lignes sont lues mot par mot et créée, puis nous affectons les arêtes à deux sommets grâce à la fonction "creeArete()" que nous avons créée dans Graphe.h puis ces arêtes sont ajoutées dans le graphe avec les sommets. La lecture de fichier est une fonction qui va renvoyer un graphe.

**III Implémentation des algorithmes vus en cours.**

**Le Parcours en Profondeur d'Abord :**

Les algorithmes de parcours en profondeur d'abord ont été décomposés en deux algorithmes :

Le premier, appelé DFS, prend en paramètre un pointeur de Sommet. Il permet d'appliquer le DFS à partir de ce sommet. Il met alors à jour les sommets du graphe en modifiant leurs valeurs "num" dans l'ordre de visite. Ainsi, les sommets ayant étés visités, et donc étant accessibles depuis le sommet placé en paramètre de fonction, verront leur variable "num" incrémentée en fonction de leur ordre de visite et les sommets non visités conserveront leur variable "num" avec une valeur de 0.

Le second parcours en profondeur ne prend rien en paramètre. Il est lui aussi appelé DFS et c'est le compilateur qui différencie les deux par la présence ou non de paramètre de fonction lors de l'appel. L'objectif ici est de parcourir tous les sommets en séparant les ensembles non accessibles depuis certain sommets. Pour cela nous parcourons tous les sommets du graphe et nous appliquons un parcours de tous leurs successeurs. La variable "num" permet toujours de déterminer l'ordre dans lequel les sommets sont parcourus et la variable "ncomp" permet de différencier les ensembles de sommets non accessibles entre eux.

**Algorithme de la Connexité entre deux sommets :**

On fait une DFS à partir du premier sommet. Si le deuxième sommet possède un numéro de parcours qui est supérieur à 0 (0 étant le numéro de base à la création d'un sommet), alors cela veut dire que le deuxième sommet a été parcouru lors de la DFS.

Alors il existe un chemin entre les deux sommets et la fonction retourne True.

Sinon cela retourne False.

**Algorithme de Ford :**

On met le poids de la source du graphe à 0.

On met ensuite les poids de tous les autres sommets à +∞.

Pour chaque sommet, on récupère ses successeurs et sur chaque successeur, on vérifie si le poids du successeur est supérieur au poids de son père + la valeur de l'arête (père, successeur), alors on met a jour le poids du successeur.

**Algorithme de dijkstra :**

Les sommets possèdent une variable boolean qui va permettre de savoir s'ils sont marqués.

Un boolean "fin" indiquera si on doit s'arrêter. Une variable va stocker le sommet où l'on se trouve. Le sommet source est stocké dans une variable.

Au départ tous les sommets sont à marqué à 'faux' et tous les poids sont à plus l'infini sauf pour la source qui est marqué à vrai et a un poids de 0.

On parcourt tant que les sommets ne sont pas tous marqués et que le boolean fin est à faux, puis on parcourt tous les sommets qui ne sont pas marqués et on test si le poids du sommet courant additionné au poids de l'arc allant au prochain sommet est inférieur au poids de ce prochain sommet.

Si c'est le cas le poids du prochain sommet prend pour valeur cette addition, et ce sommet prend comme père le sommet courant.

Puis on recherche le plus petit des poids de sommet non marqués et le sommet courant change et devient le 'prochain sommet'.

On test si le sommet courant a un poids de +l'infini, pour savoir si on s'arrête, sinon on met le marquage du sommet courant à vrai et la boucle du "tant que"

**L'algorithme de Bellman :**

Dans l'algorithme de Bellman nous mettons tous d'abords le poids de la source à 0 pour nous parcourons tous les autres sommets pour mettre leur poids à 9999999 afin de simuler +infini.

Une fois l'initialisation des poids des sommets faite nous parcourons à nouveau tous les sommets du graphe excepté la source afin d'effectuer pour chacun le parcours de leurs prédécesseurs. La récupération des prédécesseurs se fait à l'aide de la fonction "adjacencesMoins" qui retourne une liste de paires contenant un pointeur de sommet et l'arête qui le relie au sommet passé en paramètre.

Lors du parcours des prédécesseurs nous regardons si le poids du prédécesseur plus le coût de l'arête est inférieur au poids du sommet courant. Si c'est le cas le poids du sommet courant est actualisé avec cette somme.