Calculs de Cycles de Protection pour réseaux

Léo Andéol - Guillaume Chafiol - Cyril Govin - Lothair Kizardjian

UNIVERSITÉ DE MONTPELLIER

Sommaire

Tables de figures

Glossaire

CISCO: entreprise américaine spécialisé notamment dans le network management et dans la vente de matériel réseau.

.dot: extension de fichier correspondant à des fichiers de type graphes.

GNS3: logiciel permettant l'émulation ou la simulation de réseaux informatiques

Hub: Matériel réseau qui sert à amplifier un signal pour le propager.

Routeur multicast: Routeur capable de dupliquer le signal et de diffuser vers plusieurs routeurs à la fois.

Routeur optique: Routeur traitant avec un signal optique, contrairement aux routeurs classiques il nécessite aucune conversion du signal, ce qui fait qu'il est bien plus rapide par rapport à un routeur classique.

Routeur unicast: routeur capable de diffuser vers un seul routeur à la fois.

Switch: appareil réseau qui permet l'interconnexion avec une multitude d'autres appareils. Son rôle est de fractionner le réseau en domaine de collision indépendants.

WhatsupGold: logiciel de surveillance et de maintenance de réseaux.

Introduction

Dans les réseaux d'aujourd'hui, la protection des communications contre les failles est une question importante. Une technique habituelle est le calcul des routes de secours pour contourner les éléments défaillants. Grâce à l'étude des cycles et de nouveaux algorithmes, le but principal du projet est de résoudre ces problèmes de plus en plus d'actualité avec l'expansion des réseaux.

Dans ce projet nous allons résoudre le problème dont un réseau peut être victime. Il s'agit d'un incident qui va empêcher la communication de se faire entre deux points du réseau. Si ces problèmes apparaissent il faut trouver un moyen de contourner le chemin défectueux en faisant transiter les informations par un autre passage. Il faut donc trouver quel chemin de secours est le plus court et le plus optimal possible de façon à ce que les informations atteignent leur destination le plus rapidement possible tout en respectant les contraintes du réseaux que peuvent imposer les routeurs. C'est autour de ce problème d'actualité que reposera notre projet tout en approfondissant certains éléments.

Dans un ensemble général notre projet consistera en l'étude des graphes, plus précisément il s'agira de trouver et d'élaborer des fonctions dont le but sera dans un premier temps la confection de graphes en lignes de commandes. Une fois ces graphes créés il s'agira de les étudier à travers de nouvelles fonctionnalités tel que : la recherche de routes optimales entre deux points, le calcul de routes et de cycles à travers différents algorithmes, la comparaison de ces cycles/routes en durée de calcul et en efficacité et enfin, la création d'une interface graphique pour pouvoir les visualiser. Le tout en respectant des éléments contraignants tel que les routeurs unicast/multicast* et la limitation du nombre d'entrées/sorties par routeur.

Ce rapport de projet est divisé en quatre grandes parties, dans un premier temps nous allons voir l'analyse générale du projet, plus précisément, il s'agit de l'analyse de l'existant, du contexte et des besoins. Viens le rapport technique, il relate les grandes lignes de notre projet, de la conception aux résultats. La troisième partie quant à elle expose le manuel d'utilisation de notre projet, il servira pour tous ceux souhaitant utiliser notre logiciel. Enfin il sera présenté le rapport d'activité concernant principalement nos méthodes de travail durant la réalisation du projet.

1.ANALYSE

1.1 - Présentation du contexte et analyse du sujet

Actuellement il existe peu voire aucun logiciels à jour permettant de calculer des routes réseaux, et encore moins de logiciels représentants la totalité d'un réseau sous forme de graphe orienté mathématique. Notre objectif étant d'analyser un réseau (créé ou importé) et de simuler des défaillances de celui-ci afin de calculer des routes alternatives ou des cycles pour y pallier.
Évidemment le simple calcul de routes ou cycles ne suffit pas car il y a un grand nombre de choses à prendre en compte tels que le type du routeur : optique* ou non, ou encore s'il est capable de multicast ou pas par exemple.

Pour pouvoir implémenter tout ce dont nous avons besoin nous allons donc étudier ce qui existe actuellement ce qui implique les logiciels de création de graphe, les langages de programmation et les bibliothèques qu'ils proposent, les algorithmes permettant de calculer des chemins entre deux points.

Nous allons donc effectuer des sélections afin de pouvoir concevoir et réaliser notre projet. Pour ce faire l'analyse de l'existant va être le premier point qui va être abordé dans cette partie afin de pouvoir décider les outils que nous allons utiliser puis nous étudierons les besoins utilisateurs afin de pouvoir concevoir le meilleur logiciel possible.

1.2 - Analyse de l’existant

Comme énoncé précédemment il existe peu de logiciels d’analyse de réseau qui calculent des cycles et des routes dans celui-ci ou bien ce ne sont pas des logiciels ouvert au public puisque ce sont des logiciels gérant la totalité d’un réseau d’un pays. Cependant ceux qui existent (tels que GNS3* ou encore WhatsupGold*) ne proposent que de gérer un réseau de type CISCO*, ce qui ne nous intéresse pas ici car ce type de réseau est assez local c’est à dire qu’il gère tout ce qui est hub*, switch*, routeur … c’est toutefois un réseau qui peut s’étendre rapidement à grand échelle pour une entreprise. C’est cependant ce type de réseau qui est utilisé majoritairement dans le monde mais nous n’avons pas besoin de gérer tout ce qui est local car les réseaux qui vont être gérés grâce à notre logiciel seront de très grande échelle tels que celui de l’Amérique : Coronet (cf. figure n°x). Nous avons donc uniquement besoin de manipuler des routeurs et des connexions entre ces derniers ce qui est totalement différent de ce que proposent les logiciels énoncœés plus haut. De ce fait nous ne nous baserons pas sur ces logiciels et leurs fonctionnement.

Nous allons ensuite revoir les différents langages informatique afin de choisir le plus approprié à nos besoins. Par la suite les logiciels permettant de visualiser des graphes seront analysés puisque nous traiterons uniquement des graphes qui seront interprétés comme des réseaux. D’autre part il va nous falloir analyser les différentes méthodes de calculs, algorithmes qui existent et qui nous permettront d’implémenter les fonctionnalités que nous désirons. En effet il y a un très grand nombre d’algorithme de calcul de plus court chemin entre deux sommet d’un graphe. Enfin, nous allons implémenter une interface graphique, il va donc falloir soit la coder en partant de rien soit en utilisant un logiciel permettant de réaliser cela qui va dépendre du langage choisit.

1.2.1 - Analyse des langages et des bibliothèques de graphes

Actuellement, un très grand nombre de langages permet de manipuler et gérer des graphes tels que le Python, le Java, le C++ … Il va cependant falloir choisir le plus approprié en fonction de nos besoins. Pour ce qui est de nos besoins, cela sera détaillé plus tard, il nous faut un langage permettant d'optimiser le plus nos fonctionnalités. Par exemple on peut choisir entre soit un court chemin entre deux points obtenu rapidement ou bien LE plus court chemin mais obtenu moins rapidement. Les 3 langages qui se démarquent sont les 3 énoncés plus haut : Python, Java et C++.

- Python :

Python est un langage interprété assez lent mais qui implique une syntaxe légère donc il est facile de le prendre en main et d'écrire rapidement des programmes avec.
Au niveau des bibliothèques de graphe que propose Python en trouve principalement deux et qui sont open source :

-NetworkX :

Bibliothèque assez bien conçue cependant beaucoup plus lente que iGraph (environs 8 fois plus lente pour certains calculs) mais la dernière mise à jour de cette bibliothèque remonte au 30 janvier 2016 elle ne sera donc pas choisie étant donné qu'elle n'est plus tenue à jour.

-iGraph :

Comme évoqué plus haut cette bibliothèque est supérieure à NetworkX en termes de performances mais la dernière mise à jour effectuée remonte 24 juin 2015 donc pour la même raison que NetworkX nous ne l'utiliserons pas.

- Java :

Java, qui est aussi un langage interprété, est nettement plus rapide que le python mais les programmes à écrire sont plus long et plus complexes. C'est un langage un peu plus bas niveau, c'est à dire que sa syntaxe se rapproche plus de la machine que la syntaxe que nous utilisons, c'est donc un langage bien plus compliqué à comprendre que le python. Il y est donc plus facile d'optimiser des fonctions. La principale bibliothèque utilisée en Java est Jgraph qui est open source. Cette bibliothèque est à ce jour très utilisée et est basée sur JavaScript elle est très performante puisque de très grande compagnie l'utilise quotidiennement. Elle propose un composant majeur nommé mxGraph ainsi que GraphEditor qui permet d'adapter les graphes aux navigateur, c'est d'ailleurs la leader dans ce domaine et c'est son principal centre d'action.

- C++ :

C++ est une version « améliorée » du C puisqu'il est possible d'y programmer orienté objet. On y obtient donc toute la rapidité du C ainsi que sa complexité en y ajoutant les avantages de la programmation orienté objet. La principale bibliothèque graphique qui y est implémentée est la Boost

Graph Library (BGL) qui est contenu dans la bibliothèque Boost qui elle même inspire souvent la bibliothèque standard de C++ elle est donc très performante.
BGL est open source et permet de créer des graphes correspondant à nos besoins, de produire des fichiers .dot* et suis le standard POSIX (normes instaurées depuis 1990 suivie maintenant par Linux,Unix,Mac OS…).

Jgraph et BGL étaient donc les deux choix qui s'offraient à nous et notre décision finale à été de prendre BGL car Jgraph est basée sur Java et nous avons besoin d'un langage rapide et performant car nous ne pouvons pas nous permettre d'attendre 1 heure pour calculer le plus court chemin entre deux points sur un graphe très grand. D'autre part Jgraph est beaucoup plus orienté web ce qui ne nous intéresse pas du tout.
BGL nous offre tout ce dont nous avons besoin : portabilité, rapidité, efficacité, puissance et exportation de graphe dont il est possible d'obtenir une représentation visuelle par l'intermédiaire de logiciel tels que Graphviz. Ce qui de fait implique que nous utiliserons le C++ pour coder notre projet par la suite.
Nous allons donc maintenant aborder, les logiciels permettant de visualiser des graphes.

1.2.2 - Analyse des logiciels de visualisation de graphes

Pour ce qui est des logiciels permettant de visualiser des graphes en C++ on en trouve plus que 2 qui soient utilisables et surtout à jour : Tulip et Graphviz.
Tulip est un logiciel permettant de faire des visualisations en 3D grâce à des fichier importés de type GML ou TLP. Je ne développerai pas sur Tulip car la visualisation 3D ne nous intéresse pas. Cela nous amène donc à Graphviz.
Graphviz lui permet de faire des visualisations en 2D (images png/jpg ...) en important des fichiers au format .DOT qui sont ceux produits par BGL.

Notre choix s'est donc porté sur Graphviz mais au cours de l'évolution du projet nous développerons une GUI (Graphic User Interface i.e interface graphique) ce qui nous permettra de créer notre propre visualisation des graphes que nous créerons et importerons via BGL et notre logiciel.

1.2.3 – Analyse des algorithmes de calculs

Il est possible de faire de nombreux calcul avec un graphe et parmi ces calculs il y aussi un très grand nombre de possibilité de les effectuer, tous reposant sur le même principe.
Cependant tous ont des moyens de procéder différents ce qui impacte la performance et/ou la rapidité de l'algorithme en question.Il faut donc bien choisir l'algorithme qui permettra d'effectuer les calculs dont nous avons besoin avec les performances et la rapidité recquise.

Notre projet se doit de calculer des routes de secours possibles entre une source et une destination ou entre une source et plusieurs destinations ou encore trouver comment protéger un point en particulier par l'intermédiaire de cycles au sein du graphe (ce point sera développé plus tard). Nous allons donc voir les différents algorithmes qui permettent de calculer le plus court chemin, de calculer des arbres recouvrant minimaux et de calculer des cycles.

1.2.3.1 - Algorithmes du plus court chemin

Pour calculer des routes de secours entre une source et une destination l'algorithme du plus court chemin entre deux points est indispensable (ceci implique que notre graphe doit être orienté et pondéré, c'est à dire que les arêtes doivent avoir des valeurs représentant sa longueur). Et en effet il existe un grand nombre d'algorithme calculant le plus court chemin mais nous n'en présenterons ici que 2 et assez brièvement pour le moment. Les 2 algorithmes qui vont être présentés sont celui de Dijkstra ainsi que celui de Bellman-Ford.

Dans l'algorithme de Dijkstra il s'agit de construire progressivement un sous-graphe dans lequel on aura seulement les arêtes par lesquelles il faudra passer pour accéder au sommet destination depuis le sommet de départ et qui sera le plus court chemin pour y accéder.
La façon de procéder est la suivante : pour chacun des sommets voisins du sommet de départ on enregistre la distance entre ces deux sommets. Une fois tous les sommets voisins du sommet de départ parcouru on prend le premier voisin du sommet de départ et on réitère le processus. Si un chemin trouvé est plus court pour accéder à un sommet déjà visité auparavant ce sera le nouveau chemin qui sera enregistré.
La distance finale correspond à la somme des poids des arcs empruntés depuis le sommet de départ jusqu'au sommet final (si ce dernier est donné en paramètre, autrement l'algorithme calcule et renvoie le plus court chemin entre le sommet de départ et tous les sommets du graphe).

L'algorithme de Bellman-Ford lui parcourt chaque sommet du graphe et pour chacun de ces sommets il inspecte chaque arc du graphe et renvoie l'arc le plus court. Ce que cet algorithme apporte en plus par rapport à celui de Dijkstra c'est qu'il permet de prendre en compte des poids d'arc négatif dans les graphes dont il calcule le plus court chemin. Le risque des poids négatifs c'est qu'il y la possibilité que des "cycles absorbants" (e.g : lorsque l'on passe par ces cycles la distance de parcours total diminue à chaque fois qu'on le prend) de ce fait l'algorithme pourrait emprunter un nombre infini de fois ce cycle absorbant et ne rien renvoyer. C'est pourquoi il y a une partie de l'algorithme qui permet de détecter si de tels cycles existent. Cependant cela augmente énormément le temps de calcul ce qui, dans un graphe comportant un très grand nombre de sommets et d'arcs, n'est pas acceptable.

Pour la raison citée dans l'algorithme de Bellman-Ford nous ne choisirons pas ce dernier. De ce fait pour calculer les plus courts chemins dans notre application nous avons décidé de choisir l'algorithme de Dijkstra ce qui implique qu'il sera interdit d'utiliser des poids d'arc négatifs.

1.2.3.2 Algorithme de calcul d'arbre recouvrant minimal

Un arbre recouvrant minimal est un sous-graphe du graphe principal dans lequel tous les sommets du graphe principal apparaissent mais dont seulement une partie des arcs sont présents. Dans ce sous-graphe il ne doit y avoir qu'une seule façon d'accéder à un sommet, c'est pourquoi il n'y a pas tous les arcs du graphe principal. D'autre part ces moyens d'accès sont les plus court possible. Une fois toute ces conditions réunies nous avons un arbre recouvrant minimal.

Le moyen d'obtenir un arbre recouvrant minimal va reposer sur l'algorithme précédent du calcul de plus court chemin. Et comme précédemment plusieurs algorithmes existent et implémentent les plus court chemin de différentes façons. Les deux algorithmes qui vont être présentés sont celui de Prim, celui de Takahashi-Matsuyama et finalement celui de Kruskal.

Le principe de l'algorithme de Prim est très simple mais ne repose pas sur un algorithme du plus court chemin : on part d'un sommet puis à chaque étape on ajoute une arête de poids minimum adjacente à l'arbre en construction.
À l'issue de cet algorithme on obtient en effet un arbre de recouvrement minimal mais on obtient pas le plus minimal, c'est à dire qu'il existe d'autre arbres de recouvrement minimal qui pourraient être "plus" minimal. Cela est dû au fait que l'on ne prend pas en compte toute les possibilité.

On en vient donc à l'algorithme de Takahashi-Matsuyama. Celui-ci est beaucoup plus complet mais aussi plus long en temps de calcul. En effet, dans cet algorithme il faut partir d'un sommet de départ et choisir un sommet de destination dont le plus court chemin entre ces derniers est le plus court dans le graphe. Dès que la liste des sommets du plus court chemin est obtenue on applique Dijkstra entre chacun des autres sommets du graphe principal et chacun des sommets de la liste obtenue précédemment puis on ajoute à chaque fois le plus court chemin, s'il y en a déjà un on compare et on prend le plus court. Ce processus est réitéré jusqu'à ce que Dijkstra ai été appliqué à chaque sommet.
Ce procédé est donc très chronophage mais est optimal.

Pour terminer avec les arbres de recouvrement minimal l'algorithme de Kruskal est assez différent des deux derniers. En effet il ne va pas parcourir tous les sommets du graphe et chaque arc de chaque sommet mais il va seulement se concentrer sur les arcs. Cet algorithme va parcourir tout le graphe et chercher les arcs de plus faible poids. S'il y en a plusieurs avec le plus faible poids il va en choisir un arbitrairement et l'ajouter à un nouveau graphe. Il va réitérer jusqu'à avoir traité tous les arcs correspondant à ce poids (mais n'ajoutera pas les arcs qui créent un cycle) puis une fois qu'ils sont tous traités il va passer au poids minimum supérieur et recommencer. Avec cette méthode on est vraiment sûr d'obtenir le meilleur arbre de recouvrement minimal.

Maintenant que l'explication sur les algorithmes de calcul d'arbre on peut en venir aux calculs de cycles qui sont primordiaux pour l'objectif que nous souhaitons atteindre.

1.2.3.3 Algorithme de calcul de cycle

Un cycle dans un graphe est une chaîne de sommet donc le sommet du début de la chaîne et celui de la fin sont identiques. Calculer des cycles dans un graphe peut permettre de protéger des sommets. En effet si nous arrivons pour chaque sommet du graphe à trouver un cycle qui l'entoure il va être possible de rapidement pallier au problème qui de la non accessibilité de ce sommet. Cela est assez simple à comprendre : si un sommet est encadré par un cycle et que ce sommet devient inaccessible alors les communications qui passent par celui ci ne pourront plus être effective mais il sera aisé de contourner ce sommet par l'intermédiaire du cycle qui l'entoure.

Pour en venir aux algorithmes de calculs nous n'en avons pas trouvé qui ont été déjà développés en la matière de ce fait nous allons développer notre propre algorithme. Pour l'expliquer simplement il prendra en paramètre un graphe qui correspond à l'arbre de recouvrement total du graphe de base sur lequel nous allons ajouter des arêtes entre chaque feuille ce qui créera automatiquement un cycle à chaque ajout d'arête. Nous renverrons à l'issu de cet algorithme une liste de graphe dont chacun des graphes correspond à un cycle.
Une fois cette liste de graphe obtenue nous pourrons donc commencer à trouver et calculer des routes de secours lorsqu'une anomalie survient sur le réseau.

Maintenant que nous en avons fini avec l'analyse de l'existant nous allons pouvoir passer aux analyses des besoins fonctionnels et non-fonctionnels.

1.3 Analyse des besoins fonctionnels

A travers l'étude des graphes, notre projet comporte un certains nombre de fonctionnalités. Comme dit précédemment, sa principale fonction sera de calculer une route de secours entre deux points lorsqu'un problème survient sur le chemin principal entre ces deux arêtes. Mais il comporte également bien d'autres fonctionnalités tout aussi intéressante.

Une de ses première tâche, des plus importante est tout simplement la création de graphes personnalisés. Premièrement en ligne de commande mais aussi d'une manière plus ludique, l'utilisateur aura la possibilité de créer son graphe via une interface graphique. Il pourra ainsi avoir un aperçu de son graphe et de pouvoir placer les éléments à sa guise, ce qui lui rendra la tâche beaucoup plus facile et agréable.

Une fois son graphe créé, l'utilisateur pourra utiliser toute sorte de fonction dessus tel que: vérifier si son graphe est connexe, c'est à dire que tous les points sont reliés entre eux grâce aux arêtes.
Au niveau de la structure de son graphe, il pourra retrouver le chemin le plus court entre deux points, à condition qu'il existe au moins un chemin. Il pourra calculer l'arbre recouvrant le plus court, tant que son graphe est connexe. Il pourra effectuer le calcul de cycles pour n'importe quel graphe, ainsi que comparer des cycles et des routes de son graphe pour savoir lequel est le plus optimal en terme de durée de calcul et d'efficacité.

Évidemment à tout moment l'utilisateur aura la possibilité de modifier son graphe ou un graphe déjà existant. Il aura la possibilité de supprimer ou d'ajouter de nouveaux noeuds, arêtes, ainsi que de changer le type de n'importe quel noeud en le rendant unicast ou multicast grâce à une autre fonctionnalité lui donnant les informations sur chaque noeuds.

En terme de format, l'utilisateur aura la possibilité d'importer et même d'exporter ses graphes en format DOT ainsi il pourra réutiliser les graphes qu'il a créé sur une autre plateforme ou bien utiliser nos nombreuses fonctionnalités sur un graphe venant d'ailleur. (cf Annexe n°1).

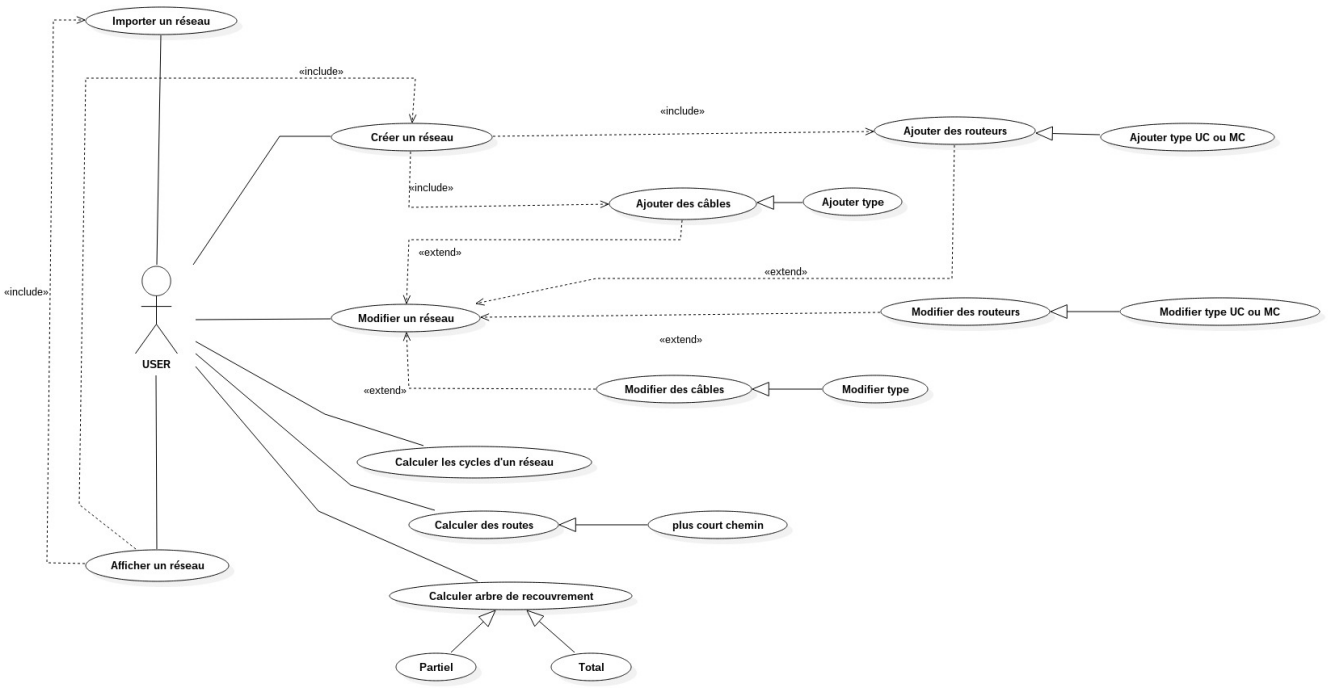

Figure 1 - Diagramme de cas d'utilisations

On peut donc à présent en venir à une conception plus détaillée de nos Networks, câbles et routeurs (cf Annexe n°2). Premièrement les routeurs : ceux-ci ne possèdent qu'un seul paramètres qui est un booléen servant à savoir s'il est multicast ou non ce qui nous permettra de faire différents traitements lors de calculs de chemins et de cycles par la suite.
Dans un second temps viennent les Câbles qui possèdent un paramètre permettant de connaître leur longueur ainsi que de deux autre paramètres qui sont des routeurs afin de connaître entre quels routeurs ils sont connectés.
Finalement la classe Network qui regroupe ces deux dernières classes et permet de créer un Graphe. Dans cette classe nous allons retrouver la plupart des fonctions qui vont être utilisées dans notre application c'est-à-dire :

- La création d'un Network via le constructeur de la classe.
- La possibilité d'ajouter des routeurs.
- La possibilité d'ajouter des câbles.
- La possibilité de retrouver un routeur du graphe via son nom.
- La possibilité de retrouver un câble du graphe via son nom.
- La possibilité de supprimer un Routeur soit par son nom soit par la passage de l'adresse du routeur en paramètre (de même pour le câble).
- La possibilité de sauvegarder le graphe ou bien d'en importer un.
- La possibilité de calculer un chemin entre deux routeurs.

- La possibilité de calculer un arbre minimum partiel du graphe via l'algorithme de Takahashi Matsuyama.
- La possibilité de calculer un arbre minimum du graphe via l'algorithme de Kruskal.
- La possibilité de calculer les différents cycles du graphe.

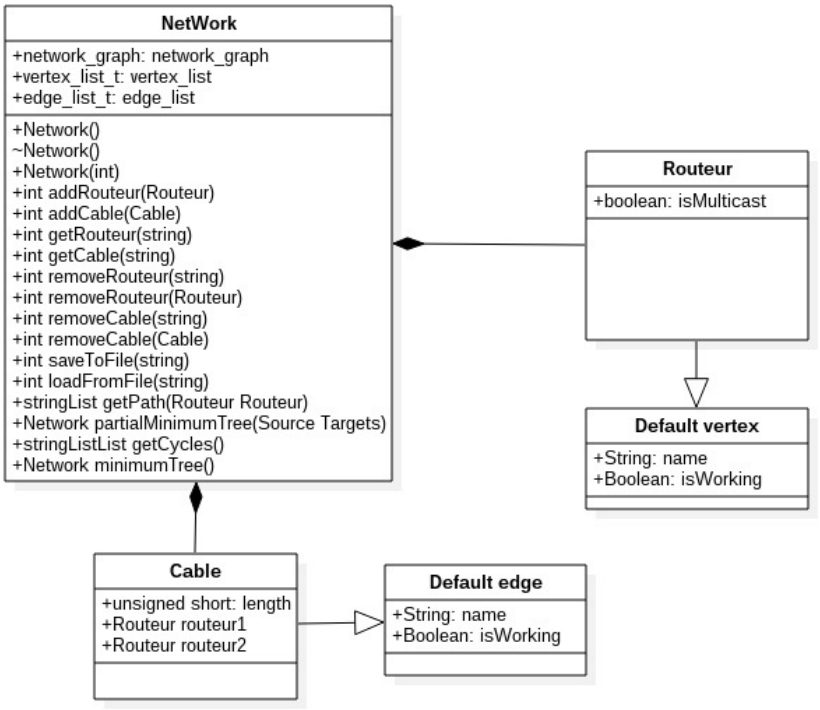

Figure 2 - Diagramme de classe

Nous avons par ailleurs fait deux diagrammes d'états transition (respectivement pour le Routeur: Annexe n°3, et le Câble : Annexe n°4) qui permettent de mieux comprendre les différents états dans lesquels ils seront lors d'une exécution du programme en fonction des actions qui seront effectuées.
Grâce à toute ces fonctionnalités, notre projet permet une étude et une gestion approfondie des graphes.

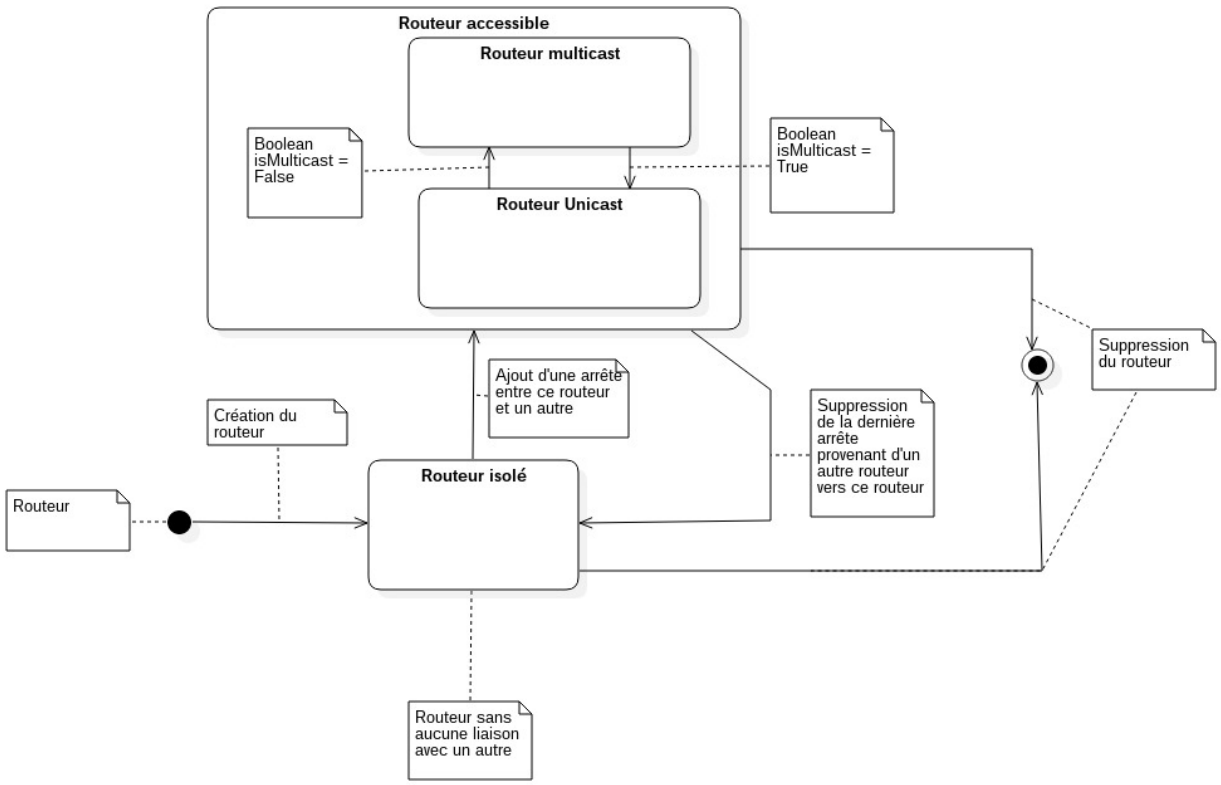

Figure 3 - Diagramme d’état transition d’un Routeur

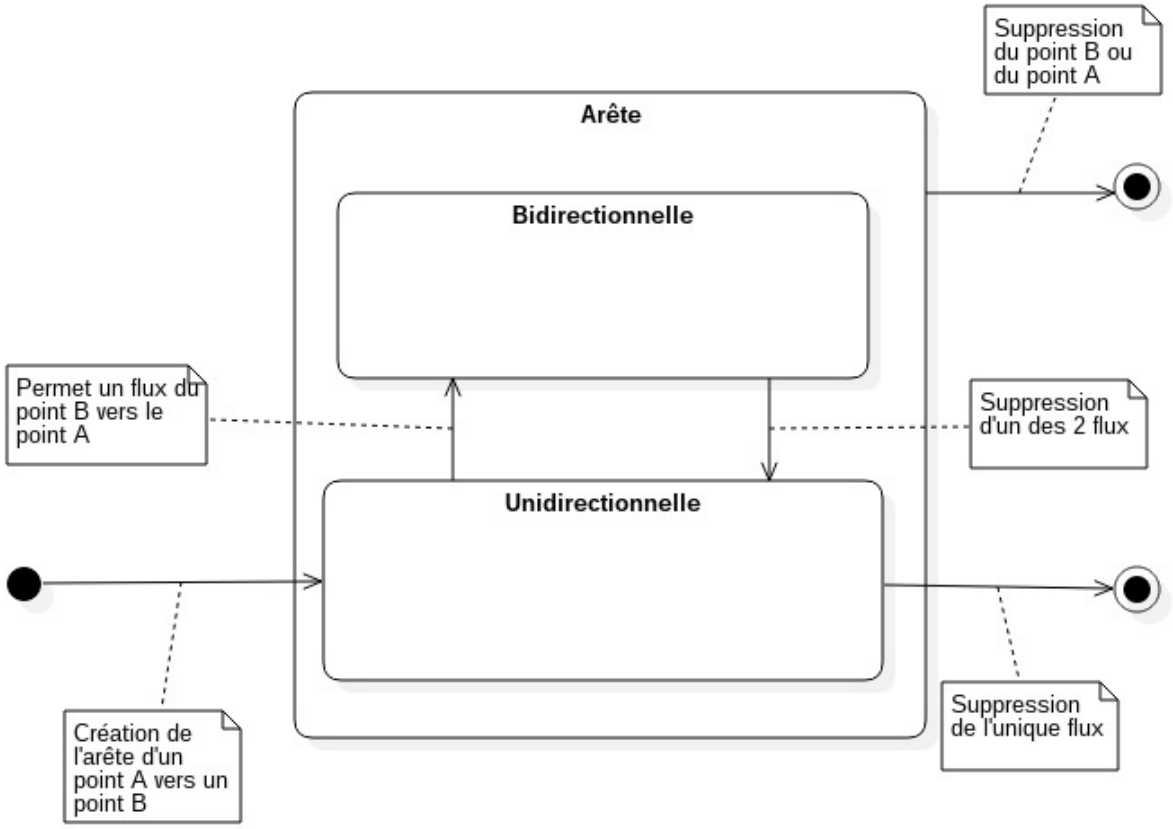

Figure 4 - Diagramme d’état transition d’un Câble

1.4 Besoins non fonctionnels

Afin de répondre à ces besoins fonctionnels, nous avons des besoins plus bas niveau, cachés à l'utilisateur, auxquels nous devons répondre.
Premièrement afin d'effectuer des calculs de chemins, d'arbres couvrants partiels ainsi que des cycles nous avons besoin d'implémenter différents algorithmes (Dijkstra, Kruskal, Takahashi-Matsuyama).

Ensuite, nous devons aussi implémenter une classe générique qui gère tous les appels de bibliothèque, et permet d'utiliser différentes structures de données, afin de simplifier l'accès aux données et leurs gestion.

Enfin nous devons créer une interface, par terminal qui permet la création, modification et application d'algorithmes sur des graphes, et ce de manière lisible et pratique, documentée.

Finalement nous devons aussi permettre des fonctions pour importer et exporter avec le format .DOT afin que les graphes soient visualisables grâce au logiciel tiers graphviz.

2. Rapport technique

2.1- Conception du projet

Nous avons commencé par élaborer des diagrammes UML afin d'organiser notre projet, nos objectifs et nos priorités.

-> On a choisi le C++ => plus rapide (recherche de rapidité)
Boost -> pour ses performances
C++ et Boost => intéressant et enrichissant
Graphviz -> Modélisation des graphes en graphique en c++
-> diagramme UML à numéroter, insérer et commenter
-> détailler les algorithmes (DOXYGEN)

2.2. Résultat

2.2.1 Introduction

???

2.2.2 Organisation de la classe Network

Afin de répondre aux attentes sur notre programme, nous avons choisi de coder la classe Network, qui est la classe principale, représentant notre réseau et donc notre graphe, de manière très générique. En effet, nous avons utilisé des structures basiques définissant les informations de base nécessaires pour tous les câbles et routeurs, et avons permis d'instancier Network donc, de manière générique en choisissant le type de structures voulues pour représenter notre réseau. Par conséquent la majorité des fonctions sont génériques ainsi que les types.

De plus, d'un point de vue plus technique, afin de simplifier l'utilisation de boost nous avons fait plusieurs fonctions de wrap, ainsi que défini des types permettant un accès plus simple aux informations du graphe. Nous avons donc utilisé des types dits dictionnaires, permettant l'associations d'une clé, souvent (et ici) hachée , et d'une valeur, afin de relier les valeurs des types représentant les sommets et arêtes de notre graphe avec des chaînes de caractères les représentant au sein de l'interface et pour l'utilisateur.

2.2.3 Creation, importation et exportation de graphes

Les fonctionnalités de base nécessaires afin de pouvoir utiliser le programme sont la création de graphes, et leur stockage. Comme annoncé précédemment, nous avons choisi le format DOT afin de stocker des graphes, les importer et les exporter avec boost ainsi que les visualiser avec graphviz.

Nous avons donc écrit des fonctions pour wrapper les appels des fonctions boost gérant l'import et l'export de fichiers. De plus, nous avons développé une interface permettant la création d'un graphe ainsi que l'application d'autres fonctions dessus, l'import et export. La création de graphe permet donc, en ligne de commande, de définir les arêtes du graphe, créant les sommets nommés par défaut; ceci effectué, la fonction renvoie un entier, l'id unique représentant le graphe créé.

2.2.4 Paramétrage de graphes

2.2.5 Calcul du plus court chemin

- Creation / importation graphes :OK
- Paramétrage graphes :OK
- Calcul PCC entre deux routeurs : OK
- Calcul arbre minimal avec Takahashi Matsuyama : OK
- Calcul arbre total avec Kruskal : TODO
- Calcul cycles : TODO
- Gérer MC / MI : TODO

2.3 Perspective de développement

Par manque de temps et dû à la complexité du projet et des outils, nous n'avons pas pu développer le projet autant que nous le souhaitions, mais nous avons toujours eu des perspectives de développement prévues.

Premièrement, nous souhaitions faire une interface graphique afin d'éviter l'utilisation de logiciels tiers pour le rendu de graphes, ainsi de faciliter la création et l'application des fonctions, et garder un maximum de clarté et de logique dans le fonctionnement de notre programme. Nous avions prévu de réaliser cette interface avec le framework Qt, car il est largement répandu, et simple d'utilisation.

De plus, nous aurions souhaité développer la possibilité d'utiliser plus de paramètres lors de la création des graphes et l'application des algorithmes sur eux mêmes, comme par exemple la prise en charge des longueurs d'ondes pour les réseaux optiques, et donc calculer les plus courts chemins par longueur d'onde.

4.Rapport d’activité

4.1 Organisation du travail

Pour organiser notre travail, nous avons choisis un modèle inspiré des méthodes agiles puisqu’il s’agit d’un principe d’auto organisation. C’est à dire que nous étions les seuls à prendre les décisions concernant le projet et ces décisions étaient prises entièrement en groupe.
Nous avons organisé notre projet sous forme de sprints, chacun durant une semaine. A la fin de chaque sprint nous prenions un rendez vous avec notre tuteur pour qu’il puisse avoir un aperçu des nouvelles fonctionnalités que nous avions créées et de l’avancement de notre travail. Ainsi il pouvait nous donner son avis sur le travail déjà réalisé, nous signaler si il y avait des petites erreurs ou des fonctionnalités qui ne correspondaient pas à ses attentes, cela nous laissait la possibilité de faire les modifications pour la prochaine rencontre. Enfin il pouvait nous aiguiller et nous montrer quelques astuces pour les sprints à venir.
Pour que chacun puisse avoir un aperçu de l’avancement des tâches durant un sprint, nous avons décidé d’utiliser la plateforme Trello, très pratique pour les travaux en groupe. Juste après chaque entretien avec notre tuteur, nous préparions la liste des tâches à réaliser pour le prochain sprint, en fonction des appréciations que nous avaient faite le professeur et des tâches prioritaire qu’il exigeait et nous avions donc une semaine pleine pour les réaliser. Une fois ces tâches rentrées sur le Trello, chacun pouvait choisir celles qu’il comptait réaliser et ainsi grâce à Trello tout le monde pouvait en être informé sans problème.

Résumé

Le sujet principal de ce projet était le calcul de cycles réseaux afin de pouvoir, en cas de problèmes survenant au niveau de routeur ou de câble, trouver un chemin de secours reliant deux routeurs. Nous avons pour cela développé une application en C++ et une de ses bibliothèques : la Boost Graph Library. Au cours de ce projet nous avons donc implémenté diverses fonctions, notamment celles qui permettent de calculer un plus court chemin entre deux points, un arbre de recouvrement minimal d'un graphe qui permet ensuite le calcul des cycles du graphe, mais aussi des fonctions d'import et d'export des graphes qui permettent de les visualiser via un logiciel nommé Graphviz. Nous avons cependant eu du mal à implémenter une interface graphique pouvant gérer l'application comme nous le voulions ainsi que l'ajout des longueurs d'ondes aux routeurs et aux câbles afin de pouvoir gérer les réseaux optiques de manière plus réaliste (principalement à cause du manque de temps). De ce que nous avons codé nous avons tout de même obtenu des résultats satisfaisant et conforme à nos objectifs.

Summary

The principal subject of this project was the computation of network cycles in order to find another path to connect two routeurs if problems concerning cables or routeurs would occur. Thus we developped an application in C++ using one of its library : the BGL (Boost Graph Library). During this project we implemented various functions, especially the ones that compute the shortests paths between two points, a minimum covering tree of a graph which eventually led to calculate the graph's cycles, but also functions that can import and export graphs which can bee seen through another app called Graphviz. However we have had troubles implementing a graphic interface that would manage the application as we wanted but also to add wavelengths to the routeurs and cables to manage optical networks on a more realistic way (mainly because we lacked time). From what we coded we still had satisfiying results compliant to our objectives.