RAPPORT DE PROJET SIMULATION

ALGORITHME GHS

Essossolam Jordy Aquiteme
UNIVERSITE DE REIMS CHAMPAGNE-ARDENNE | 2019-2020

Professeur : Thibault Bernard

Table des matières

INTRODUCTION 2
LE CHOIX DE L'IMPLEMENTATION 2
L'ORGANISATION MINIMALE 2
FONCTIONEMENT 4
Les composants de la Simulation 4
L'exécution globale 4
La description de l'algorithme 5
EXEMPLE D'EXECUTION 7
CONCLUSION 12

INTRODUCTION

La simulation d'un algorithme est un processus permettant de vérifier sa fiabilité et aussi son fonctionnement vis-à-vis des données en entrée. Lors d'une simulation, il peut être aussi intéressant dans certains cas, de rechercher les faiblesses.

Dans cette optique, nous avons réalisé une simulation de l'algorithme Gallagher Humblet Spira (GHS) dans le cadre du cours RT0803 du master Réseau Télécommunication de l'université de Reims Champagne-Ardenne. GHS est un algorithme de « spanning tree », c'est-à-dire un algorithme d'arbres couvrants basés sur l'envoi de messages asynchrones. Nous disposons déjà de l'algorithme, mais là le but est de pouvoir le traduire dans un langage de programmation au choix et d'en observer le fonctionnement.

Dans ce rapport, nous allons présenter en premier lieu le choix de notre implémentation, ensuite le fonctionnement et pour finir les exemples d'exécutions et les résultats obtenus.

LE CHOIX DE L'IMPLEMENTATION

Afin de mener à bien ce projet, nous avons décidé d'implémenter la simulation de l'algorithme avec le langage Java.

Nous avons utilisé le langage Java, tout d'abord parce que c'est le langage que nous maîtrisons le plus. Ensuite, Java est aujourd'hui un langage dont la rapidité d'exécution est comparable à ceux de bas niveau comme le c et le c++.

Afin de faciliter le développement et le débogage, nous avons privilégié l'utilisation de l'IDE NetBeans et la version Java 8.

L'ORGANISATION MINIMALE

Dans le cadre de ce projet, nous avons décidé de garder une implémentation logique plus près de la théorie. Alors, nous avons utilisé le paradigme de la programmation orientée objet (POO) pour représenter les éléments comme le montre la figure ci-dessous.

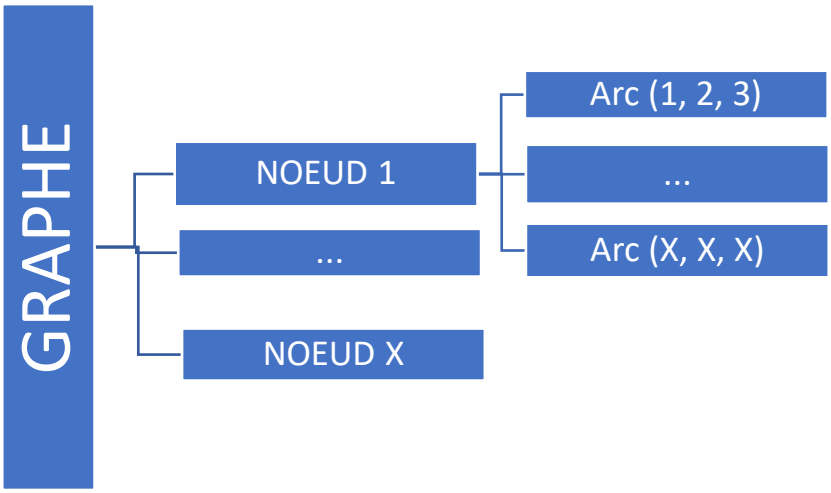


Figure 1 Arborescence d'un graphe.

L'OBJET GRAPHE

L'objet graphe permet de représenter le graphe en entré fourni par l'utilisateur. C'est est une matrice au format fichier texte (.txt) et ordonné comme suit :

Nœud a	Nœud b	Poids
1	2	3
1	3	4
...	...	...
X	X	X

Le caractère de séparation est la virgule («, »).

Ex : (1, 2, 3)

À l'intérieur de cet objet, se trouvent les codes qui permettent de piloter notre simulation et de l'initialiser.

L'OBJET NODE (Nœud)

L'objet « node » sert à représenter le nœud d'un graphe. Principalement, le nœud a au moins :

- Un identifiant de nœuds.
- Et une liste d'adjacences (Arc).

Et aussi les variables nécessaires au déroulement de l'algorithme GHS.

En situation réelle, le nœud peut être considéré comme un site. Donc tout naturellement, c'est cet objet qui contient les différents blocs de l'algorithme GHS.

L'OBJET ARC (Adjacence)

L'objet « arc » ou arête permet de représenter une adjacence qui est une liaison entre deux nœuds. Évidemment, un nœud peut avoir plusieurs arcs.

L'arc contient au minimum :

- L'identifiant des deux nœuds.
- Et dans notre cas le poids de cette liaison.

FONCTIONEMENT

Les composants de la Simulation

Threads

Le but de ce projet étant la simulation, nous avons utilisé les threads pour simuler l'exécution des tâches de l'algorithme GHS. Ces threads fonctionnent de manière indépendante comme dans la réalité. Un thread est propre à un nœud.

File de messages

Comme expliquer lors des séances de Visio, nous avons implémenté une file de messages (BlockingQueue de Java) qui permet de regrouper tous les messages. Là encore, toujours dans l'optique de garder une logique, nous avons représenté un message sous forme d'objets.

La file de messages est asynchrone pour tous les nœuds et synchrone pour les messages à destination du même nœud.

Message

L'objet message comporte au moins :

- L'identifiant de l'émetteur et du destinataire.
- Et le contenu du message.

Optionnellement, certains messages ont en plus les paramètres.

Étant donné, le contenu du message correspond à un bloc de l'algorithme GHS, nous l'avons remplacé par un entier qui correspond au bloc concerné (classe Bloc dans le code source). Ceci, pour rendre l'exécution plus rapide.

L'exécution globale

Comme mentionné un peu plus tôt dans ce rapport, l'exécution de la simulation requiert en entrée, un fichier texte représentant la matrice du graphe.

Etape 1 : initialisation du graphe

Après avoir fourni le fichier, l'exécution démarre avec l'initialisation de ce graphe. Le code écrit permet de créer un graphe avec ses nœuds et arc (adjacences).

Etape 2 : initialisation de la file de messages et des threads

Le graphe initialisé, l'exécution continue avec l'initialisation de la file de message et des threads (pour chaque nœud créé). Ensuite, l'algorithme de GHS est lancé.

Pour donner l'ordre à nos différents threads (nœuds) de démarrer leurs processus d'initialisation de l'algorithme GHS, nous avons créé un message avec pour émetteur « -1 » pour représenter l'identifiant du système. Alors, ces messages, sont ajoutés également lors de l'initialisation de la file de messages.

Etape 3 : Récupération, traitement et envoie de messages sur chaque nœud.

Les threads ne sont pas démarrés dans l'ordre. Ceci permet d'avoir une exécution aléatoire.

Une fois lancés, ceux-ci récupèrent les messages qui leur sont destinés, les traites et envoient des messages dans la file de messages si nécessaire, jusqu'à la fin du déroulement (lorsque les messages « TERMINÉ » sont envoyés).

Afin de suivre le déroulement, certains messages comme la réception et l'envoi de messages sont affichés à l'utilisateur dans la console. Pour savoir également le nombre de messages envoyés et reçus, nous avons également mis en place des compteurs spéciaux.

Les messages de type « traiter le message plus tard » sont simplement remis dans la file de message. Mais cela peut augmenter considérablement le nombre de messages reçus par le nœud si l'état de celui-ci ne change pas. Pour des raisons de visibilité, nous avons choisi de ne pas afficher ce type de message.

Etape 4 : Fin de l'exécution

À la fin de l'exécution, les nœuds affichent leurs différents états. Tous canaux de chaque site placé à « branch », sont celles qui font partie de l'arbre couvrant.

La description de l'algorithme

Tout d'abord, il est à noter que pour que l'algorithme puisse fonctionner, le graphe en entrée doit être pondéré.

Lexique :

- Basic : état d'une adjacence qui n'est ni « branch » ni « reject ».
- Branch : état d'une adjacence faisant partie de l'arbre couvrant.
- Reject : état d'une adjacence exclut de l'arbre couvrant.

Au niveau 0 (« initialisation ») :

- Chaque nœud choisit une adjacence de poids minimum entre toutes ses adjacences et la marque comme « branch ».
- Ensuite, il envoie un message (connect) à travers cette adjacence.
- Et attend la réception d'un message.

Broadcast (« initiate »)

C'est un message de diffusion envoyé aux canaux « branch » à l'exception du nœud dont il reçoit le message. Il contient l'identifiant du nœud et son niveau.

Convergence

Pour que l'algorithme converge, les nœuds de même fragment (c'est-à-dire de même niveau) s'échangent différents messages ou exécutent des procédures internes en envoyant leur poids minimum et leur niveau. Ces messages sont « test », « TEST », « accept », « reject », « report » et « REPORT ».

Changement de racine

C'est un message envoyé entre deux nœuds pour propager un meilleur chemin qu'ils ont récemment reçu.

Lors de l'exécution de l'algorithme, il se passe également des mécanismes d'absorption ou de fusion.

Le mécanisme d'absorption se produit lorsqu'un nœud avec un niveau inférieur hérite du niveau de celui avec un niveau supérieur.

Le mécanisme de fusion se produit lorsque deux nœuds partagent la même adjacence de poids minimum. Si le niveau du nœud 1 est le même que celui du nœud 2, ils incrémentent (+1) tous deux leur niveau.

EXEMPLE D'EXECUTION

Avant tout, il faudrait commencer par renseigner le chemin des fichiers comme le montre l'image ci-dessous :

```
String chemin = "C:\\Users\\DELL\\Documents\\Java Projects\\rt0803\\src\\";
String filename = "grapheEx1.txt";
```

Etat départ du graphe

Après avoir lancé l'exécution de la simulation, le code affiche l'état de départ du graphe fourni comme suit :

```
==============NOEUD 1==============
CANAUX
{2=basic, 4=basic}
SUCCESSEURS
[{id=2: poids=4.0}
{id=4: poids=3.0}
]
NIVEAU
0
MCAN
0
PERE
0
MESSAGES ENVOYES
0
MESSAGES RECUS
0
=================FIN==============,
```

Affichage des Messages

Afin de distinguer les différents messages, nous avons décidé de les afficher avec différentes couleurs.

```
2 start initialisation
2 send connect(0) to 4
1 start initialisation
4 start initialisation
1 send connect(0) to 4
3 start initialisation
4 send connect(0) to 2
3 send connect(0) to 2
4 recieve connect from 2 Niv = 0
4 send initiate(1, 1.0, find) to 2
4 recieve connect from 1 Niv = 0
4 recieve connect from 1 Niv = 0
4 recieve connect from 1 Niv = 0
```

Figure 2 Extrait d'affichage des messages avec différentes couleurs.

Etat final du graphe

Lorsque l'algorithme aura convergé et que tous les messages auront été envoyé, chaque nœud affichera son état comme suit :

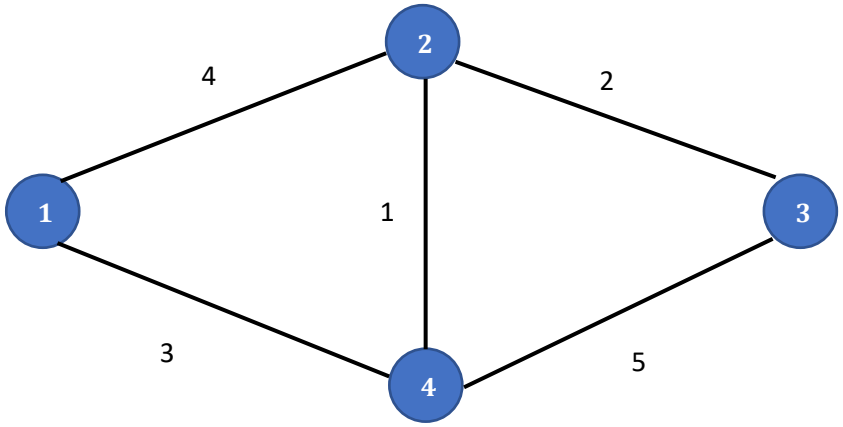
```
==============NOEUD 2==============
CANAUX
{1=basic, 3=branch, 4=branch}
SUCCESSEURS
[{id=1: poids=4.0}
{id=3: poids=2.0}
{id=4: poids=1.0}
]
NIVEAU
1
MCAN
-1
PERE
4
MESSAGES ENVOYES
3
MESSAGES RECUS
6
==================FIN==============
```

Dans certains cas, le nombre de messages reçus peut être grand à cause du mécanisme « traiter le message plus tard ». En effet, lorsqu'un nœud remet un message dans la file et lorsque ce nœud également n'a aucun message en attente de traitement, ce dernier passera son temps à répéter le mécanisme jusqu'à ce que son état ne change.

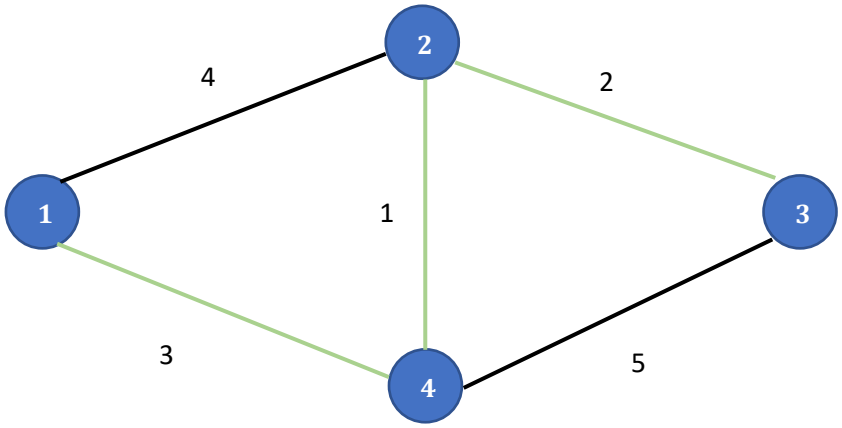
Graphes vus au cours

Graphe 1

DEPART



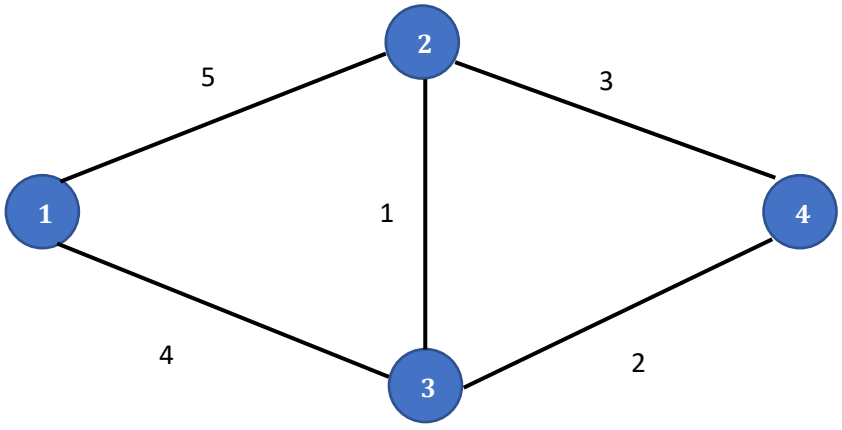
A LA FIN DE LA SIMULATION



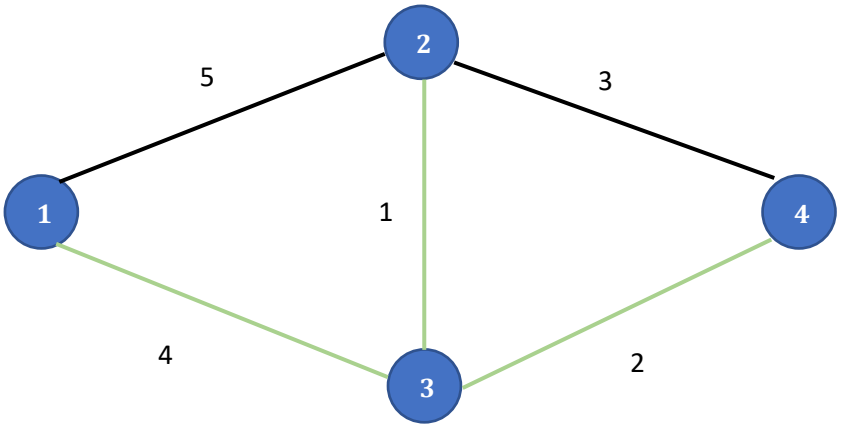
Les traits marqués en vert représentent l'arbre couvrant. Cet arbre a été tracé, avec les informations fournit par chaque nœud à la fin de la simulation (canaux « branch ») de chaque nœud.

Graphe 2

DEPART



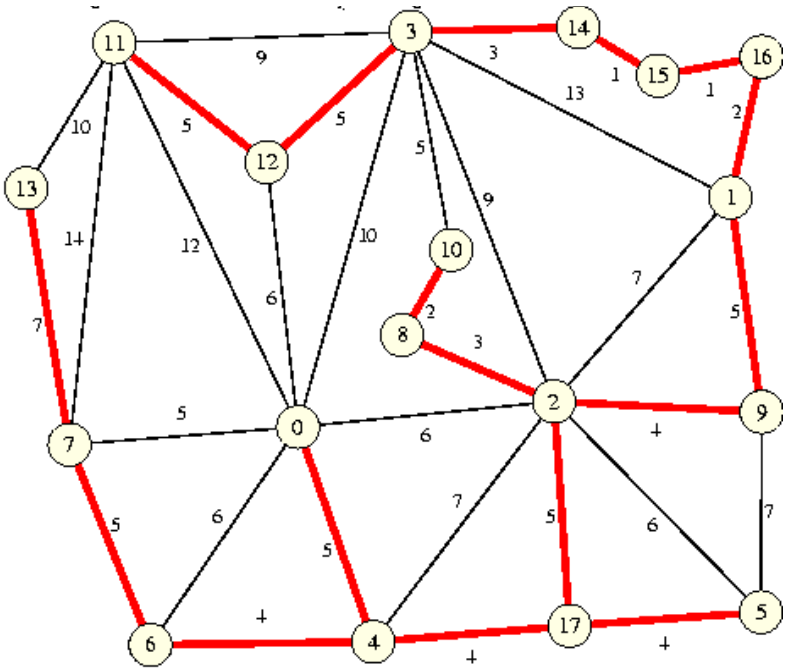
A LA FIN DE LA SIMULATION



Autres graphes

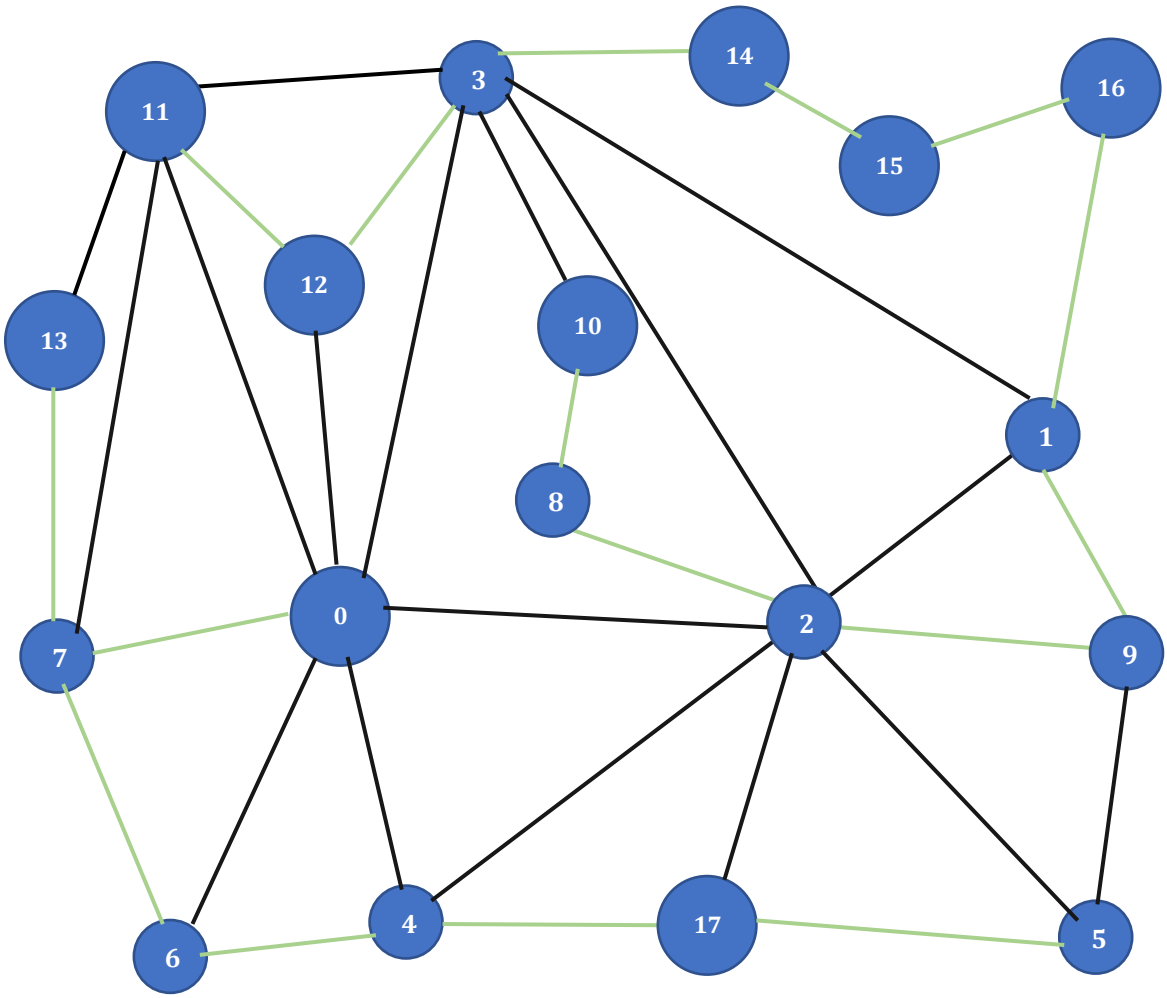
Afin de tester l'efficacité de notre implémentation, nous avons décidé également de soumettre certains graphes trouvés sur internet.

Ce graphe est composé de 17 nœuds.



A LA FIN DE LA SIMULATION

À la fin de la simulation, nous avons une exécution similaire à différence que le nœud 7 a été choisi par notre simulation à défaut du nœud 4. Ces deux nœuds ont le même poids.



REMARQUE :

Étant donné que nous avons utilisé les Threads, nous avons ajouté quelque temps de pause pour permettre d'avoir un bon affichage des éléments à la fin de la simulation.

CONCLUSION

Cela a été une grande opportunité d'avoir implémenté une simulation d'exécution d'un algorithme dans le cadre du cours RT0803. Cela nous a permis de mieux comprendre le fonctionnement de l'algorithme GHS pour ma part.

Au départ nous avons eu quelques difficultés à comprendre le travail demandé, mais avec quelques éclaircissements, nous avons réussi.

Finalement, l'implémentation marche plutôt bien avec la plupart des graphes avec lesquels nous avons fait nos tests.