Rapport 1ère partie Projet
HAI721I - Programmation répartie

Département informatique

Jérémie BENTOLILA 21906253,
Maxime BOURRET 21901688

22 novembre 2022

Table des matières

1 Introduction

1.1 Présentation du sujet

Ce projet est réalisé dans le cadre du module HAI721I et a pour objectif de résoudre le problème de la coloration de sommets. Celui-ci est organisé en deux étapes, nous traiterons dans ce rapport de la première. Vous pouvez retrouver le code source de cette dernière sur GitHub.

L'objectif de cette partie est de développer une application pour la mise en place d'un graphe à colorier. Ce graphe se traduit par un réseau de processus interconnectés en fonction d'une description donnée en entrée. Cette description est définie sous le format des instances données à l'adresse : http://cedric.cnam.fr/~porumbed/graphs/.

L'application doit donc permettre de produire n'importe quelle instance suivant ce format. Chaque instance est représentée de la manière suivante : un premier caractère (c, p, ou e) de la ligne représente le contenu de cette dernière comme suit :

```
c <commentaire>
p <nombre_nœuds> <nombre_aretes>
e <nœud1> <nœud2>
```

En effet, "c" désigne les lignes de commentaire, "p" se trouve une seule fois dans chaque fichier et sert à décrire le nombre de nœuds et d'arêtes que contient le graphe. Quant à "e", il désigne une arête entre 2 nœuds.

1.2 Exécution

Afin de faire fonctionner notre projet, ouvrez un terminal et exécutez les commandes suivantes :

```
git clone https://github.com/MqxBrt/HAI721I-Projet
cd HAI721I-Projet
make run
```

Un `Makefile` est inclus dans les fichiers téléchargés et permet donc de compiler et de lancer le projet en une commande. Si vous souhaitez simplement le compiler, exécutez la commande `make compile`. Vous devez ensuite fournir un nom de fichier contenu dans le dossier *files*. Ce dernier doit contenir un graphe décrit avec la structure évoquée dans la partie 1.1 ci-dessus. Le programme va ensuite s'exécuter en lançant le serveur sur le port 10 000 par défaut.

2 Conception

Afin de réaliser la première partie du projet, nous avons mis au point une trame conceptuelle répertoriant toutes les actions à faire pour nos acteurs. L'ordre des actions s'effectue de haut en bas :

Noeud.c	**Script.py**	**Serveur.c**
	Lancement du script Python Récupération du nom du fichier à traiter Vérification du nombre d'arêtes et estimation des délais	
		Lancement du parseur sur le fichier Récupération du tableau des voisins de chaque nœud Création d'une socket d'écoute des nœuds
	Lancement des n clients	
Création des sockets Passage de la socket serveur en mode écoute Connexion au serveur		
		Pour les n nœuds : Acceptation du nœud Envoi du nombre de voisins
Récupération de son nombre de voisins Passage de la socket voisin en mode écoute Transmission de l'IP et du port de la socket d'écoute serveur Transmission de l'IP et du port de la socket d'écoute voisin		
		Récupération des IPs + ports et stockage dans des tableaux à la case i
Déconnexion du serveur		
		Fermeture de la socket d'écoute *Lorsque les n nœuds ont été traités, pour les n nœuds :* Connexion aux nœud
Acceptation du serveur		
		Transmission des adresses des voisins
Récupération de ses voisins		
		Déconnexion du nœud *Lorsque les n nœuds ont été traités* **FIN**
Fermeture de la socket serveur Création d'une socket de connexion pour chaque voisin Connexion à tous ses voisins Acceptation des connexions de tous ses voisins **COLORATION**		

FIGURE 1 – *Trame d'exécution du projet*

3 Implémentation

Nous avons choisi de réaliser une implémentation type client-serveur afin d'interconnecter les sommets entre eux. Pour automatiser l'exécution, nous avons réalisé un script en `Python` qui va d'abord récupérer le nom du fichier à traiter, puis lancer le serveur après avoir vérifié que le fichier ne contenait pas plus d'arêtes que de port autorisés et enfin qui va lancer les nœuds. Cela nous permet de ne pas avoir à lancer des centaines de nœuds manuellement. Les commandes de base sont :

```
./bin/serveur numero_port nom_fichier
./bin/noeud 127.0.0.1 numero_port id_noeud
```

Afin de pouvoir les exécuter en parallèle dans le même terminal, nous avons rajouté `"&"` à la fin de chaque commande pour que leur appel ne soit pas bloquant pour la console.

Nous avons également créé le fichier *commun.h* qui contient des fonctions génériques qui facilitent la manipulation des sockets (création, nommage, connexion, envoi, réception...) et sont utilisées dans les fichiers *serveur.c* et *noeud.c*.

Pour une meilleure lisibilité, nous avons ajouté de la coloration lors des messages affichés dans la console, ainsi nous obtenons le code couleur suivant :

- **Orange** : message affiché par l'exécution du parseur
- **Violet** : message affiché par l'exécution du serveur
- **Bleu** : message affiché par l'exécution d'un nœud
- **Vert** : message de succès affiché par un nœud lorsqu'il est connecté à tous ses voisins
- **Rouge** : message d'erreur affiché avant l'arrêt d'un processus

3.1 Parseur

La première chose que nous avons réalisé pour le projet est le parseur en C. Il est contenu dans le fichier *parseur.h* et est appelé par le serveur au début de son exécution. Après avoir compris la forme générale des fichiers contenant les graphes, nous avons décidé que le parseur devait renvoyer un tableau de tableaux où chaque sous-tableau à la case i correspond aux voisins du sommet i. Par exemple, l'arête `"e 1 2"` donne le tableau : `[ [1], [0] ]` (les numéros sont réduits de 1 car les tableaux commencent à l'indice 0).

Or, les graphes peuvent être parfois complexes et les sommets n'ont pas forcément tous le même nombre d'arêtes. Ainsi, afin de manipuler facilement ces données, nous avons fait en sorte que le parseur renvoie une structure comme suit :

```
struct graphe {
    int ** tab_voisins;
    int * tab_degres;
```

```
    int nombre_sommets;
};
```

L'attribut `nombre_sommets` est, comme son nom l'indique, le nombre de sommets du graphe et sert à connaître la taille des deux autres tableaux de la structure. `tab_voisins` correspond donc au tableau de tableaux décrit plus haut et `tab_degres` est un tableau d'entiers où le nombre contenu à la case i équivaut au degré du sommet i et sert donc à connaître la taille sur sous-tableau à la case i de `tab_voisins`. C'est cette structure qui contient toutes les informations du graphe qui est transmise au serveur.

3.2 Serveur central

Une fois la structure récupérée, le serveur connaît donc le graphe et va se mettre en attente des nœuds. Lorsqu'un nouveau nœud se connecte à lui, il va lui envoyer son nombre de voisins (`tab_degres[i]` où i correspond au nombre de nœuds déjà traités). Le serveur va ensuite recevoir deux adresses et ports qu'il va stocker dans deux tableaux différents, toujours à la case i. La première, celle à laquelle il va recontacter le nœud une fois que tous les sommets se seront connectés à lui, sera stockée dans `adNoeudsContact[i]`. La deuxième, celle à laquelle les voisins pourront contacter le nœud, sera stockée dans `adNoeudsVoisins[i]`.

Lorsque tous les sommets se sont connectés une fois et ont échangé leurs informations, le serveur détient toutes les ressources nécessaires à la création du graphe : les relations de voisins de par le résultat du parseur et les adresses sur lesquelles vont se connecter les processus de par sa communication avec les processus nœuds. Il va recontacter chacun d'eux dans l'ordre où ils sont arrivés afin de leur envoyer les adresses où ils pourront se connecter à leurs voisins. Une fois cela fait, le travail d'interconnexion du serveur est terminé, le processus s'arrête.

3.3 Nœuds

Les nœuds sont créés par le script `Python` et un `id` leur est donné avec comme seul but de les différencier dans les messages sur le terminal. Chacun commence par créer et nommer trois sockets :

- `socket_connexion_serveur` : sert à se connecter au serveur afin de récupérer le nombre de voisins et d'envoyer les adresses et ports des deux sockets suivantes
- `socket_ecoute_serveur` : sert à recevoir les adresses des voisins une fois que le serveur recontacte les nœuds
- `socket_ecoute_voisins` : sert à écouter tous les voisins qui vont se connecter puis être acceptés

Le nœud va ensuite mettre en écoute `socket_ecoute_serveur` puisque la longueur de la file d'attente est déjà connue (un seul serveur donc 1). Il se connecte au serveur,

récupère son nombre de voisins, passe `socket_ecoute_voisins` en mode écoute maintenant que la longueur de la file d'attente est connue puis envoie les adresses et ports de `socket_ecoute_serveur` et `socket_ecoute_voisins` grâce à la fonction `getsockname(ds, (struct sockaddr *) &ad, &len)`. Nosu avons choisi de faire cela car le nommage des sockets est automatique afin d'éviter les problèmes tels que "*Port already in use*".

Après avoir été déconnecté du serveur, il attend que ce dernier le recontacte pour lui envoyer les adresses de ses voisins. Elles seront stockées dans le tableau `adressesVoisins`. Lorsqu'elles sont toutes reçues, le serveur est déconnecté et un tableau de socket de la taille du nombre de voisins est créé où chacune envoie une demande de connexion à un voisin. Enfin, les voisins sont tous acceptés avec une boucle `for` et des `accept` avant d'afficher le message de confirmation de l'interconnexion.

3.4 Difficultés rencontrées

Au delà des erreurs liées à des erreurs de syntaxe dans notre programme, nous avons rencontré quelques problèmes importants lors du développement qui nous ont fait modifier notre implémentation. Le premier rencontré, qui a été géré assez facilement, était dans le parseur à propos de la manipulation du tableau de tableaux. Finalement, la structure que nous avons présenté plus haut a parfaitement collé avec ce que nous souhaitions obtenir en sortie du parseur.

La plus grosse difficulté a été rencontrée au moment de tester notre projet sur un grand nombre de sommets. En effet, lors de nos tests pendant l'implémentation, nous n'avions pas encore réalisé le script `Python` et utilisions un anneau de 3 sommets. Or, lorsque nous avons pour la première fois testé avec des centaines de nœuds, notre ancienne version a montré ses limites. Jusqu'à présent chaque nœud prenait en paramètre un port et exécutait ses sockets sur `port + [0; nombre_voisins + 2]`. Ainsi le serveur récupérait simplement l'adresse et le port du client lors de sa première connexion et incrémentait de 1 et de 2 le port pour savoir où recontacter le nœud. Lors de l'exécution, le script `Python` récupérait le nombre de voisin du sommet actuel afin de savoir sur quel port lancer le nœud suivant. Sur des graphes plus conséquents que l'anneau de taille 3, nous nous sommes donc retrouvé avec le problème de "*Port already in use*". C'est pourquoi nous avons laissé le système nommer automatiquement les sockets sans se soucier du port (sauf pour le serveur) et avons fait en sorte que les nœuds récupèrent l'adresse et le port pour l'envoyer au serveur (comme expliqué plus haut avec la fonction `getsockname(ds, (struct sockaddr *) &ad, &len)`.

4 Conclusion

Cette première partie de projet a permis de nous familiariser davantage avec le réseau dans le langage C. Nous avons pour mettre en application nos connaissances dans ce domaine sur un projet concret en obtenant des résultats visibles et satisfaisants.

Malgré les difficultés rencontrées, nous avons réussi à mettre en place un réseau de processus correspondant à un graphe qui fonctionne jusqu'à environ 65 000 arêtes, nous allons maintenant poursuivre le projet vers la deuxième étape : la coloration du graphe.