VERSMEE Erwan Groupe 3131

KERMANI Benjamin 29/01/2020

Compte Rendu TP 4:

Analyse de logs apache

[**I. Spécifications**](#_6xxmxes9e2mc) **2**

[I.1 Spécifications générales](#_yemo5i1tc34p) 2

[I.2 Spécifications détaillées](#_b8v6db2axuc) 2

[I.3 Tests réalisés](#_ay7omlf69hh2) 3

[**II. Architecture globale**](#_xae2w9oblus3) **4**

[**III. Choix des structures de données**](#_zfub0opi8et) **5**

[III.1 Affichage des 10 adresses les plus consultés](#_djjiqvnpn2jb) 5

[III.2 Graphe des interactions](#_q3r9lqux51jv) 6

# I. Spécifications

## I.1 Spécifications générales

Le programme a pour objectif de réaliser 2 tâches sur un fichier de logs Apache :

* Fournir la liste des 10 documents les plus consultés, avec pour chacun son adresse et son nombre d’accès affiché dans le terminal. On pourra rajouter certaines options pour supprimer certains résultats
* De fournir, dans le cas où la syntaxe et la sémantique des options respecte le cahier des charges, un graphe des “interactions”, i.e les nombres d’accès, entre ce que nous appellerons “les référents”, les adresses depuis lesquelles on effectue un accès ,et “les cibles”, les adresses que l’on souhaitent accéder.

## I.2 Spécifications détaillées

Dans cette partie nous expliciterons quels sont les tests unitaires que nous avons implémentés et quels sont les résultats nous attendons.

Nous rappelons la description de chaques options :

* **(pas d’options) “<nom\_du\_fichier>.log” :** Fournit la liste des 10 documents de “<nom\_du\_fichier>.log” les plus consultés avec leurs nombres d’accès dans le terminal
* **“-g <nom\_du\_graphe.dot> <nom\_du\_fichier>.log” :** Fournit la liste des 10 documents les plus consultés ET fournit le graphe des accès entre les différentes adresses
* **“-t heure <nom\_du\_fichier>.log”:**  Prend en compte les accès contenu dans le créneau horaire [heure , heure+1] ou l’heure représente l’heure locale.
* **“-e <nom\_du\_fichier>.log”:** Exclut les logs contenant certaines extentions (image, css, JS,...) pour l’adresse cible.
* **“-s statut <nom\_du\_fichier>.log”:** Retient seulement les requêtes ayant un statut égal à celui demandé en paramètre

On rappelle également que ces options sont **cumulables** et que pour obtenir le .png nous devons effectuer la commande : “dot -Tpng -o nom\_fichier.png nom\_fichier.dot”

Attention, il faut rentrer **d’abord** les extensions, et en **dernier** le nom du fichier log

## I.3 Tests réalisés

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Numéro de test** | **Options** | **Description du test** | **Résultat attendue** | **Statut de retour** |
| 01 | X | Plus de 10 résultats ayant le plus de consultations  (des résultats ont le même nombre de consultations) | Affiche les 10 adresses les plus consultés dans le terminal | 0 |
| 02 | X | Le fichier de log est vide | Message informant l’utilisateur que le fichier est vide | 1 |
| 03 | X | Fichier dont l’extension n’est pas la bonne | Message d’erreur, et réécrire syntaxe | 1 |
| 04 | X | Fichier qui n’existe pas | Message d’erreur, et réécrire syntaxe | 1 |
| 05 | -g  <nom\_du\_Graphe.dot> | Fournit le graphe du log correspondant | Affichage des 10 résultats dans le terminal et .dot créé | 0 |
| 06 | -t heure | L’heure est compris entre [0;24] | Effectue le traitement normalement | 0 |
| 07 | -t heure | L’heure n’est pas comprise entre [0;24] | message d’erreur, et réécrire syntaxe | 1 |
| 08 | -e | Présence de fichier avec extension du type image, css ou js | Effectue le traitement | 0 |
| 09 | -e | Le fichier de log ne contient aucun fichier image,css,... | Le traitement est inutile mais renvoie le résultat attendu | 0 |
| 10 | -s | On ne conserve que un statut particulier rentré après l’option -s | Le traitement effectué exclut tous les statuts différents de celui rentré | 0 |
| 11 | -s | Aucun statut rentré après le -s | Renvoie un message d’erreur | 1 |
| 12 | -x | On utilise une option inconnue | Le programme renvoie un message et effectue le traitement normalement | 0 |

On considère que la dernière entrée de la ligne de commande est le nom du log et que les options qui précèdent cette valeur peuvent être ordonnées de manière aléatoire. Attention toutefois, pour le “-g”, le nom du graphe avec la bonne extension doit suivre immédiatement, de même que l’heure recherchée doit suivre immédiatement le “-t” et le statut le “-s”.

# II. Architecture globale

Afin de réaliser notre application, nous avons implémenter 3 classes .cpp et leurs interfaces .h correspondant (et un main)

La première classe, Log.cpp nous permet d’identifier ce qu’est un log en récupérant les données intéressantes du fichier .log et en les stockant dans une structure “InfosLog” prévue à cet effet (structure passée en attribut de la classe Log). Ici, les données qui nous intéressent seront l’heure, le statut, l’adresse référente, l’adresse cible et son extension si elle existe de la requête examinée. Le fichier log sera ouvert dans le constructeur et fermé dans le destructeur de la classe Log.

Cette encapsulation est bénéfique au programme et à sa “réusabilité”. En effet le fait de lire un fichier de log dans une classe à part entière en récupérant plus d’informations que celle dont on a besoin dans notre cas, nous permettra de pouvoir réutiliser cette classe en adaptant la structure des données en fonction du traitement que l’on souhaite effectuer sans avoir à changer la manière dont on lit/récupère les données d’un log.

La deuxième classe, DixSites va nous permettre de trouver les dix adresses cibles les plus recherchées dans le fichier log : un premier classement sera effectué dans une unordered\_map avec pour clé l’adresse cible, et pour second paramètre le nombre de requêtes effectuées vers cette adresse cible. La multimap servira à “inverser” la unordered\_map pour récupérer comme clé le nombre de requêtes effectuées vers l’adresse cible et trier les résultats en fonction de ce nombre ; en cherchant les 10 derniers éléments de la multimap, on obtient les dix documents les plus consultés. Le troisième attribut servira lorsque l’option -e est demandée par l’utilisateur : le programme va comparer l’extension de l’adresse cible avec les éléments du vecteur extension pour savoir si il faut insérer cette adresse cible dans la unordered\_map ou pas.

Enfin, la troisième classe Graphe permettra de créer un fichier Digraph qui pourra par la suite être transformé en image. Dans le unordered\_set seront stocké les noeuds du graphe : toutes les adresses (cibles et référentes), tandis que dans la map, on stockera les valeurs d’arcs (le nombre d’interactions entre chaque adresse, s’il en existe).

Le vecteur extension a la même fonction que dans la classe DixSites. La méthode Afficher() permet d’afficher une version textuelle du graphe. Les deux méthodes Ajouter() des classes DixSites et Graphe sont appelées suivant que la ligne de commande comporte des options -e ou -t (Ajouter(Log &,int,bool)) ou pas (Ajouter(Log &). La méthode testVide() de DixSites permet de tester si la unordered\_map est vide après insertion du fichier log.

# III. Choix des structures de données

## III.1 Affichage des 10 adresses les plus consultés

Nous avons décidé d’utiliser une “table de hachage” : pour un premier usage, nous n’avons pas besoin d’un résultat trié, il nous suffit de récupérer les adresses cibles et le nombre de requête effectué vers ces adresses. Ainsi, une table de hachage permet d’avoir de meilleurs performances d’insertion et de recherche qu’une autre structure de donnée qui effectuerait un tri :

Hash\_table <Adresse\_cible,Nombre\_de\_requêtes>

Elle aura donc pour clé l’adresse cible et pour valeur le nombre d’accès demandé sur cette même adresse.

La seconde structure de donnée : un arbre binaire de recherche (arbre rouge et noire) nous servira à “inverser” la première structure de donnée ; ses performances sont généralement moins bonnes qu’une table de hachage, mais cette structure permet d’effectuer un traitement nécessaire sur nos données, un tri. Par ailleurs, on a besoin qu’une même clé puisse être utilisée pour deux adresses différentes(clés multiples), par exemple lorsqu’on a un même nombre de requête (hits) pour deux adresses différentes :

BST <Nombre\_de\_requêtes,Adresse\_cible>

Nous afficherons ainsi les 10 résultats contenant le plus de requêtes (par ordre décroissant).

En terme de STL nous utiliserons une **unordered\_map puis une multimap.**

Pour notre première map (unordered\_map), la complexité de la recherche est en général de O(1) (dans le meilleur des cas) mais peut atteindre O(n) (cela dépend de la fonction de hachage utilisée) ; l’insertion se fait en O(1).

Concernant la multimap, l’insertion (seul opération effectuée sur la multimap par notre opération) se fait en O(log(n)) ce qui assure une complexité temporelle raisonnable.

(Note : Pour avoir une petite idée, le jeu de données entier contient 100 000 logs, on a donc log(100 000) = 5)

## 

## III.2 Graphe des interactions

Pour stocker les noeuds, on introduira dans notre structure toutes les adresses (référentes ou cibles) différentes. Encore une fois nous n’avons pas besoin d’un résultat trié ; nous utiliserons donc un ensemble non trié (qui est basé sur un arbre rouge et noire dans la STL)

Hash\_table<Adresse>

Pour les valeurs des arcs, on utilisera un arbre ayant pour clé un couple d’adresse (clés uniques) et comme valeur stockée le nombre de requête effectuée de la première adresse de la clé (référente) vers la seconde (cible). L’implémentation par table de hachage étant dans ce cas trop compliquée, nous nous sommes tournés vers un arbre binaire de recherche :

BST <<Adresse\_référente,Adresse\_cible>,Nombre\_de\_requêtes>

Nous créons ainsi un fichier digraph en récupérant les labels des noeuds dans la Hash map, et les labels des arcs dans le BST.

En terme de STL, la première structure de donnée correspond à **un unordered\_set, et la seconde à une map.**

L’insertion dans un unordered\_set se fait en temps constant.

Quant à la map, l’insertion et la recherche se font en O(log(n)).