# Compte-rendu C++ TP n°4

Charles Javerliat, Fabien Narboux
Binôme 3128
INSA Lyon, 3-IF

23 janvier 2020

# Table des matières

**1 Introduction** **3**

**2 Spécifications de l'application** **3**
2.1 Spécifications générales . . . 3
2.1.1 Résultats attendus du programme . . . 3
2.2 Spécifications détaillées . . . 4
2.2.1 Spécifications des URLs . . . 4
2.2.2 Spécifications du filtrage des fichiers médias . . . 4
2.2.3 Spécifications du filtrage par créneau horaire . . . 4
2.2.4 Spécifications du top 10 . . . 4
2.2.5 Spécifications de la génération du graphe . . . 4

**3 Architecture de l'application** **5**
3.1 Choix réalisés . . . 5
3.2 Diagramme de classe . . . 5

**4 Choix des structures de données** **5**
4.1 Objectifs . . . 5
4.2 Structures de données choisies . . . 5
4.2.1 Conteneur associatif <URL → Compteur d'accès> . . . 5
4.2.2 Conteneur associatif <(URL d'origine, URL cible) → Compteur d'accès> . . . 6
4.2.3 Liste triée du classement . . . 6
4.3 Pseudo-code et complexités algorithmiques . . . 8

**5 Manuel d'utilisation** **9**

# 1 Introduction

L'objectif de ce TP est de réaliser une application permettant l'analyse de fichiers de logs produit par un serveur Apache. L'application devra être capable d'afficher un top 10 des URL les plus visitées, ainsi que de générer un graphique GraphViz-Dot afin de visualiser le trafic.

# 2 Spécifications de l'application

## 2.1 Spécifications générales

Notre application doit être capable de réaliser deux tâches précises :
— Afficher le top 10 des URL les plus visitées
— Générer un graphique du trafic entre les différentes URL
Pour ce faire, la commande d'éxecution doit respecter le format suivant :

```
./bin/analog [-e] [-t heure] [-g nomGraphe.dot] <fichierLogs.log|fichierLogs.txt>
```

Nous avons fait le choix d'être strict sur l'ordre des options, ainsi, il n'est pas possible d'avoir l'option `-e` qui viendrait après `-t`, par exemple. Cela simplifie l'implémentation.

*Pour voir la liste des options et leur utilité, se référer à la partie Manuel d'utilisation.*

### 2.1.1 Résultats attendus du programme

Selon les options utilisées dans la commande d'exécution du programme, nous pouvons obtenir différents résultats. Nous avons donc fait un tableau récapitulatif des différents cas possibles.

| Commande exécutée | Résultat attendu | Test associé |
|---|---|---|
| ./analog /tmp/anonyme.log | Affichage dans la console du top 10 | Test n°1 |
| ./bin/analog -e /tmp/anonyme.log | Affichage dans la console du top 10<br>Exclut les fichiers javascript, images, ... | Test n°2 |
| ./bin/analog -t 12 /tmp/anonyme.log | Affichage dans la console du top 10<br>Ne considère que les URL visitée entre 12h et 13h (exclu) | Test n°3 |
| ./bin/analog -g graph.dot /tmp/anonyme.log | Affichage dans la console du top 10<br>Génère un graphe du trafic dans le fichier graph.dot | Test n°4 |
| ./bin/analog -e -t 12 /tmp/anonyme.log | Affichage dans la console du top 10<br>Exclut les fichiers javascript, images, ...<br>Ne considère que les URL visitée entre 12h et 13h (exclu) | Test n°5 |
| ./bin/analog -e -t 12 -g graph.dot /tmp/anonyme.log | Affichage dans la console du top 10<br>Exclut les fichiers javascript, images, ...<br>Ne considère que les URL visitée entre 12h et 13h (exclu)<br>Génère un graphe du trafic dans le fichier graph.dot | Test n°6 |
| ./bin/analog -t 12 -g graph.dot /tmp/anonyme.log | Affichage dans la console du top 10<br>Ne considère que les URL visitée entre 12h et 13h (exclu)<br>Génère un graphe du trafic dans le fichier graph.dot | Test n°7 |
| ./bin/analog -e -g graph.dot /tmp/anonyme.log | Affichage dans la console du top 10<br>Exclut les fichiers javascript, images, ...<br>Génère un graphe du trafic dans le fichier graph.dot | Test n°8 |

Tout autre commande exécutée ne respectant pas l'un des format ci-dessus renverra une erreur, sous la forme d'une aide indiquant le format correct à utiliser pour lancer le programme. Nous avons listé quelques exemples types levants cette erreur ci-dessous.

| Commande exécutée | Résultat attendu | Test associé |
|---|---|---|
| ./bin/analog | Erreur syntaxe : ./bin/analog [-e] [-t heure] [-g nomGraphe.dot] <fichierLogs.log\|fichierLogs.txt> | Test n°9 |
| ./bin/analog fichierLogs.log aaa bbb | | Test n°10 |
| ./bin/analog fichiersLogs.dot | | Test n°11 |
| ./bin/analog -g | | Test n°12 |
| ./bin/analog -g nomGraphe.abc fichierLogs.log | | Test n°13 |
| ./bin/analog -g -e nomGraphe.dot fichierLogs.log | | Test n°14 |
| ./bin/analog -g nomGraphe1.dot -g nomGraphe2.dot fichierLogs.log | | Test n°15 |
| ./bin/analog -t 24 fichierLogs.log | | Test n°16 |
| ./bin/analog -t abc fichierLogs.log | | Test n°17 |
| ./bin/analog -t 22.4 fichierLogs.log | | Test n°18 |
| ./bin/analog -g nomGraphe.dot fichierLogs.log -e | | Test n°19 |

*Pour pouvoir lancer les tests, il est important de lancer la commande make au préalable afin que le fichier binaire soit généré.*

## 2.2 Spécifications détaillées

Nous avons fait certains choix au sein de notre programme qu'il est nécessaire de lister ici. C'est notamment le cas pour le traitement des URL qui demandaient réflexion quant à la façon de les lire et les intérpréter.

### 2.2.1 Spécifications des URLs

- On retire des adresses locales la partie de l'adresse du serveur des logs, c'est à dire *http ://intranet-if.insa-lyon.fr*
- Les adresses pointant vers des dossiers existants sous deux formes possibles (ex : */temps/* ou */temps*), nous avons fait le choix de les uniformiser en retirant les / de fin d'adresse.
- Certaines adresses contenant des paramètres passés après un délimiteur *'?'*, nous avons fait le choix de les uniformiser en ne gardant que la partie précédent *'?'*. Par exemple, l'adresse */index.php ?param=val* est équivalente à */index.php*.
- Certaines adresses ne possèdent pas d'URL d'origine, par exemple si on y accède via un marque-page, une application externe. L'adresse d'origine est alors marquée *'-'*. Nous avons fait le choix de conserver un noeud de nom *'-'* dans le graphe.
- Nous ne prenons pas en compte le code de retour de la page dans le comptage du nombre d'accès à une URL.

### 2.2.2 Spécifications du filtrage des fichiers médias

L'option `-e` permet de filtrer les URL pour ne pas compter celles correspondants à des fichiers médias (javascript, images, ...). Ainsi, nous avons choisi de filtrer les types suivants : *JPG, JPEG, jpg, jpeg, bmp, PNG, png, gif, ico, css, js*.

### 2.2.3 Spécifications du filtrage par créneau horaire

L'utilisateur doit nécessairement rentrer un entier naturel $heure \in [\![0; 24[\![$ correspondant au créneau $[heure; heure+1[$ des logs conservées. Par défaut, si aucun créneau n'est précisé, aucun filtre sur l'horaire n'est appliqué.

### 2.2.4 Spécifications du top 10

Le top 10 est affiché dans tous les cas, il affiche les URL les plus visitées, en les triant d'abord par nombre d'accès, puis par ordre alphabétique en cas d'égalité.

### 2.2.5 Spécifications de la génération du graphe

Un graphe est généré avec l'utilisation de l'option `-g nomGraphe.dot`, il contient l'ensemble des noeuds correspondants aux différents URLs et représente le trafic à l'aide de flèches pondérées par le nombre d'accès depuis une origine vers cette URL. Comme précisé précédemment, les noeuds affichés respectent les filtres, et nous affichons également le noeud pour un accès à une URL via une origine inconnue.

# 3 Architecture de l'application

## 3.1 Choix réalisés

Tout en gardant en tête que l'application peut-être agrandie, améliorée, nous avons décidé de séparer au maximum les modules. Par conséquent nous avons une classe par tâche à réaliser, pour respecter le principe de responsabilité unique. Nous avons donc :

- La classe **LecteurLog** qui s'occupe de lire le fichier de logs, nettoyer les URL, et de remplir les structures de données nécessaires au reste du programme.
- La classe **LogData** qui s'occupe de contenir les données.
- La classe **CompteurAccesURLAbsolu** qui permet de compter le nombre d'accès à une URL depuis n'importe quelle origine.
- La classe **CompteurAccesURLRelatif** qui permet de compter le nombre d'accès à une URL depuis une URL d'origine donnée.
- La classe **GenerateurGraphe** qui s'occupe de générer le fichier .dot du graphique à partir des structures de données appropriées.
- La classe **AfficheurClassement** qui s'occupe d'afficher le classement des n premières URL (ici n = 10 mais cela pourrait être une autre valeur).

Le fait de découpler les classes de génération du graphique et d'affichage du classement nous permet de rajouter facilement des modules si on voudrait de nouvelles fonctionnalités d'analyse des données.

## 3.2 Diagramme de classe

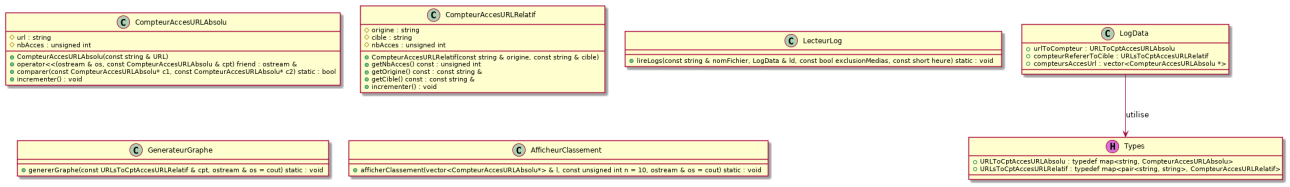


FIGURE 1 – Diagramme de classe de l'application

# 4 Choix des structures de données

## 4.1 Objectifs

Les fichiers de logs pouvant atteindre rapidement quelques giga-octets en taille, il est important de choisir des structures de données pertinentes qui répondent à nos besoins. Notre application doit être efficace à la lecture, à l'insertion de données, mais également au traitement, donc à la lecture et l'interprétation de ces dernières.

## 4.2 Structures de données choisies

En vue des tâches à réaliser (graphe + top 10) nous sommes partis sur l'utilisation de trois structures de données ayant chacune une tâche spécifique.

### 4.2.1 Conteneur associatif <URL → Compteur d'accès>

Nous avons utilisé un conteneur associatif pour lier une URL à un compteur de nombre d'accès absolus (depuis n'importe quelle URL d'origine/referer), nous avons donc défini un type customisé qui s'écrit comme suit **map<string, CompteurAccesURLAbsolu>**. La map nous permet d'y insérer une valeur en complexité $\mathcal{O}(\log n)$. En effet, pour rajouter une valeur dans l'arbre représentant la map (Cf. figure ci-dessous) on parcourt ses branches de manière dichotomique pour trouver où placer le noeud, comme on divise par deux le nombre de possibilité à chaque embranchement, on a bien cette complexité logarithmique. La complexité est identique (même principe pour la recherche que l'insertion) pour récupérer une valeur associée à une clé ($\mathcal{O}(\log n)$). Son implémentation est donc réalisée sous la forme d'un arbre rouge-noire qui permet un équilibrage parfait et une recherche dichotomique efficace (pour chaque noeud, le noeud fils de gauche est inférieur, celui de droite est supérieur). Le schéma d'une telle structure dans notre cas se représenterait sous cette forme :

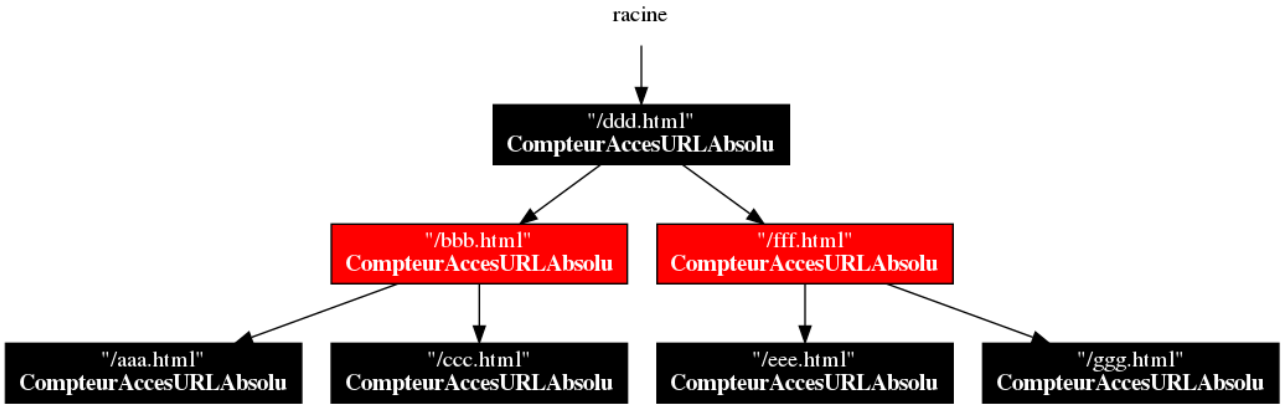


FIGURE 2 – Schéma de la structure sur un exemple

### 4.2.2 Conteneur associatif <(URL d'origine, URL cible) → Compteur d'accès>

Nous avons utilisé un conteneur associatif pour lier une URL à un compteur de nombre d'accès relatif (depuis une URL d'origine/referer donnée), nous avons donc défini un type customisé qui s'écrit comme suit **map<pair<string, string>, CompteurAccesURLRelatif>**. La complexité est similaire à la structure citée avant soit $\mathcal{O}(\log n)$ en insertion et en recherche. Le schéma d'une telle structure dans notre cas se représenterait sous cette forme :

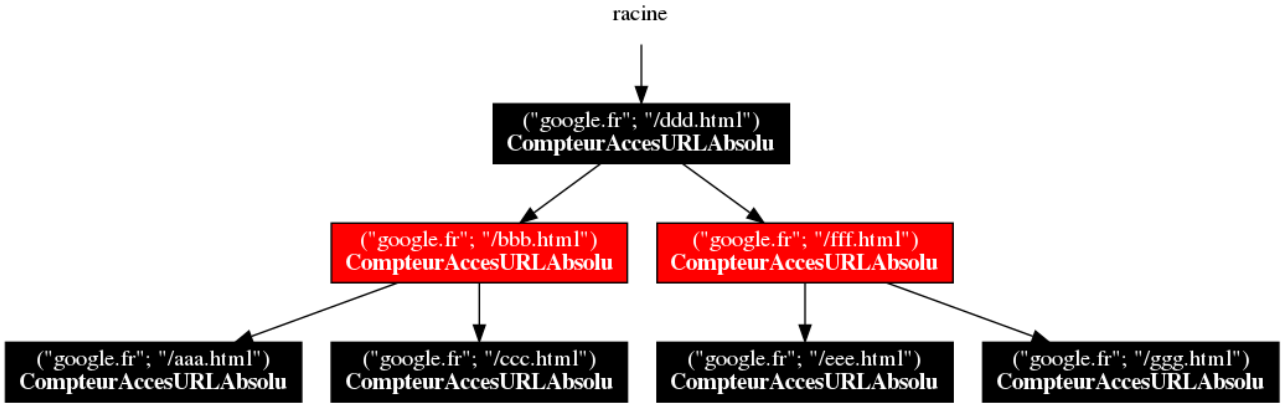


FIGURE 3 – Schéma de la structure sur un exemple

### 4.2.3 Liste triée du classement

Nous étions d'abord parti du principe que nous utiliserions un set contenant les instances de CompteurAccesURLAbsolu, triés selon le nombre d'accès. Toutefois, après lecture de la documentation, nous nous sommes rendu compte qu'un set ne se trie pas quand on met à jour les attributs des objets qu'il contient. Or, notre algorithme met à jour la variable comptant le nombre d'accès. Il nous fallait donc trouver un moyen de pouvoir trier la liste à la fin de l'algorithme d'insertion ou au début de l'algorithme d'affichage du classement, c'est donc ce que nous avons fait. Nous sommes donc parti sur l'utilisation d'un **vector<CompteurAccesURLAbsolu*>**, qui contient donc les références aux compteurs pour éviter la redondance d'informations en mémoire, et nous trions cette liste via l'algorithme d'introspective-sort de complexité $\mathcal{O}(n \log n)$. Cet algorithme est un dérivé du quicksort (tri par pivot). Le cout de l'insertion dans un vector est de $\mathcal{O}(1)$ dans le cas où celui-ci n'a pas besoin d'être redimensionné en mémoire. En mémoire, le vector se représente sous la forme d'un tableau, comme un array, mais il peut évoluer dynamiquement (il double sa taille dès qu'il est trop petit). Le schéma d'une telle structure dans notre cas se représenterait sous cette forme :

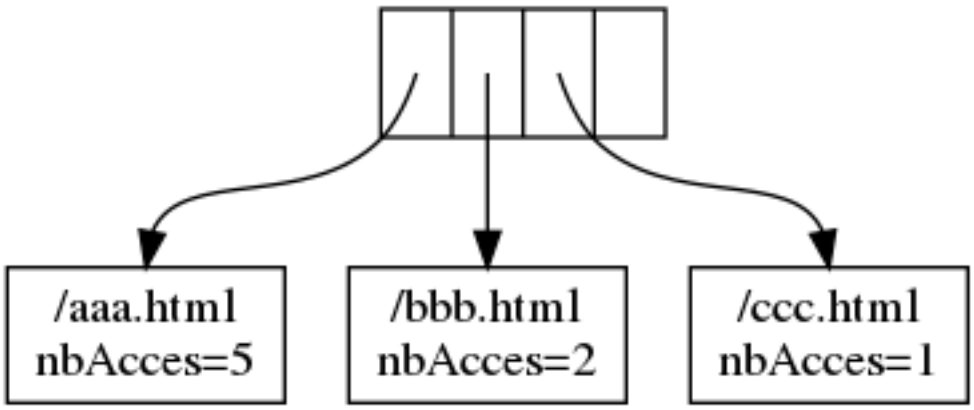


FIGURE 4 – Schéma de la structure sur un exemple

## 4.3 Pseudo-code et complexités algorithmiques

Pour pouvoir tester l'efficacité des structures de données utilisées, nous avons écrit le pseudo-code des différentes parties de l'application avant de les coder en C++.

### Algorithme d'insertion des données

```
map<string, CompteurAccesURLAbsolu> cptAccesAbsolu;
map<pair<string,  string>, CompteurAccesURLRelatif> cptAccesRelatif;
vector<CompteurAccesURLAbsolu> classement;

pour chaque ligne du fichier logs O(n)
        si l'url cible n'a jamais été rencontrée O(log n)
                crée un CompteurAccesURLAbsolu
                l'ajoute à cptAccesAbsolu O(log n)
                l'ajoute à classement O(1)
        si la paire (origine; cible) n'a jamais été rencontrée O(log n)
                crée la paire (origine; cible)
                l'ajoute à cptAccesRelatif O(log n)
        incremente le nombre d'accès absolus de la cible O(1)
        incremente le nombre d'accès relatifs de la paire (origine; cible) O(1)
```

Dans le cas de cet algortihme on trouve donc, dans le pire des cas, une complexité de $\mathcal{O}(4n \log n + 2n) \simeq \mathcal{O}(n \log n)$

### Algorithme d'affichage du top 10

```
vector<CompteurAccesURLAbsolu> classement;

trie classement par nbAcces et ordre alphabétique O(n log n)

pour i allant de 0 à 9 O(10)
        cpt ← classement[i]  O(1)
        affiche cpt.url et cpt.nbAcces
```

Dans le cas de cet algortihme on trouve donc, dans le pire des cas, une complexité de $\mathcal{O}(n \log n + 10) \simeq \mathcal{O}(n \log n)$

### Algorithme de génération du graphique

```
map<pair<string,  string>, CompteurAccesURLRelatif> cptAccesRelatif;
map<string, unsigned int> noeudsToID;

pour chaque tuple de cptAccesRelatif O(n)
        si l'url d'origine ne se trouve pas dans noeudsToID O log(n)
            ajoute le noeud d'url origine au graphe
            insère l'url du noeud et son ID dans noeudsToID O log(n)
        si l'url de cible ne se trouve pas dans les noeudsToID O log(n)
            ajoute le noeud d'url cible au graphe
            insère l'url du noeud et son ID dans noeudsToID O log(n)
        ajoute une flèche origine -> cible de label CompteurAccesURLRelatif.nbAcces
```

Dans le cas de cet algortihme on trouve donc, dans le pire des cas, une complexité de $\mathcal{O}(4n \log n) \simeq \mathcal{O}(n \log n)$

# 5 Manuel d'utilisation

**NOM**

analog - analyseur de logs

**USAGE**

analog [-e] [-t heure] [-g nomGraphe.dot] <fichierLogs.log|fichierLogs.txt>

**DESCRIPTION**

Analyse des logs Apache, permet de générer un classement des URL les plus visitées, ainsi qu'un graphe de trafic.

**OPTIONS**

**-e**

Ne prend pas en compte les médias (images, javascript, ...)

**-t** heure

Filtre les logs par l'horaire d'accès à l'URL ($heure \in [\![0; 24[\![$)

**-g** nomGraphe.dot

Génère un grahique dans le fichier nomGraphe.dot