TP4 C++

Table des matières :

1. Introduction ……………………………………………………………………………………………………..
2. Spécifications ……………………………………………………………………………………………………..
3. Architecture Globale ……………………………………………………………………………………………
4. Structure des Données ……………………………………………………………………………………………
5. Conclusion ………………………………………………………………………………………………………………
6. Introduction

Lors de ce TP, nous avons essayé d’appliquer des modules de la Standard Library C++ et aussi de manipuler des flux de données pour concevoir une application. Nous avons essayé de développer des solutions génériques et réutilisables, qui peuvent facilement être adaptées pour des nouveaux problèmes.

Plus spécifiquement, notre application devrait lire un fichier journal qui contenait des logs Apache (des lignes décrivant des “parcours” entre des pages Web), les modéliser et les ranger dans une structure de données qui nous permettrait de les manipuler facilement. Étant donné que la taille du fichier peut être très volumineuse, le choix d’une structure de données cohérente devient crucial.

Ce projet a été fait en deux parties principales : la conception des classes et la reflexion aux structures de données qui seraient utilisées pour mieux répondre à la problématique et l’implementation en langage C++.

Niveau conception, le principal défi a été le choix de la Structure des Données. Nous avons essayé de réfléchir aux principales opérations réalisées, et le type de relation entre les éléments pour guider notre choix, ceci sera expliqué plus profondément dans le parties qui suivent.

Une fois que la conception était faite et le projet état bien défini, l’implementation en langage C++ n’a pas été très difficile.

ll. Spécifications

Le programme fonctionne en ligne de commande. L’installation est faite en suivant les instructions dans le fichier README.md. Nous pouvons découper le fonctionnement général en trois parties :

Main:

Point d’entrée du programme : réalise la lecture du fichier et le traitement initial ( filtrage ou pas ) des logs.

Cas normaux :

On peut mettre les options dans n'importe quelle ordre (comme une commande linux classique)

* + Options possibles :
  + -g nomFichier.dot : génère un fichier GraphViz
  + -e : exclut documents de type image, CSS ou JS.
  + -t heure : filtre sur les heures [heure, heure+1[
  + [default] Afficher les 10 documents les plus consultés

Cas limites :

* + Si nomFichier.dot est déjà existant : on demande confirmation à l’utilisateur
  + Option inconnue : warning
  + Absence d'argument nom de fichier : afficher l’aide

Cas d'erreurs :

* + Le paramètre t doit être un entier entre 0 et 23.
* Heure et nomFichier.dot sont des arguments obligatoires, leur absence implique une erreur.

LogAnalyzer:

Manipulation de la structure des données.

Cas normaux :

* + Pour chaque ligne de log, on ajoute la requête à la structure de données
  + On doit pouvoir exclure tous les documents images/css/javascript
  + Chaque requête est identifiée par le triplet (Referer; Cible;DateTime)

Cas limites :

* + La requête existe déjà : on ajoute une nouvelle fois la requête : la structure de données doit donc supporter les doublons

Cas d'erreurs :

* + Ligne de log mal formée : n'est pas censé arriver.

GraphViz gen:

Transformation de la structure des données en un graphe.

Cas normaux :

* + Pouvoir passer de la structure de données des logs analysées à un graphe bien formé GraphViz

Cas limites :

* + Pas de requêtes à exporter: générer un graphe vide (à définir selon syntaxe GraphViz)

lll. Architecture Globale de l’Application

Diagramme de Classes


* Request :

Chaque ligne de log sera transformée dans un objet de type Request. Il contient tous les attributs nécessaires ainsi que les opérateurs surchargés pour la lecture et écriture.

* LogFileParser :

Cette classe fait la lecture du fichier en appliquant les options demandées et les filtres.

* HourFilter et ExtensionFilter :

Filtres implémentés à partir de la classe abstraite AbstractFilter. HourFilter peut être utilisé pour faire un filtre de [heure,heure+1[ ou [heureDebut, heureFin[ et ExtensionFilter peut être utilisé pour filtrer n’importe quelle extension. Nous avons choisi de laisser les extensions en paramètre pour pouvoir réutiliser les classes.

* Caches :

Afin de gagner en espace mémoire, chaque string URI/referer est associée à un identifiant entier unique. Ainsi le graphe manipule des entiers qui sont moins coûteux en mémoire et ne récupère la valeur de la chaînes de caractères que si nécessaires (affichage ou export).

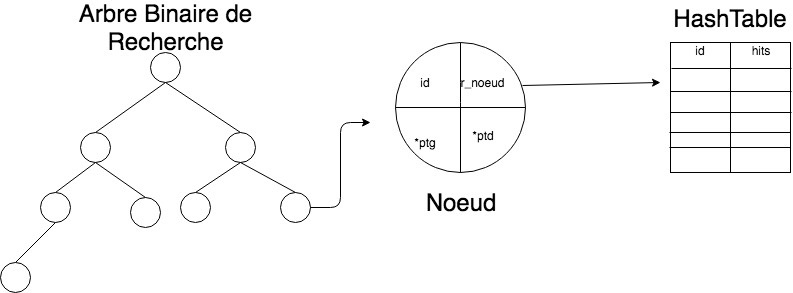
* ResourceNode

Cette classe représente les noeuds de notre graphe. Elle contient l’identifiant de la cible et le nombre de consultations ( ici appelés “hits”).

* DirectedGraph:

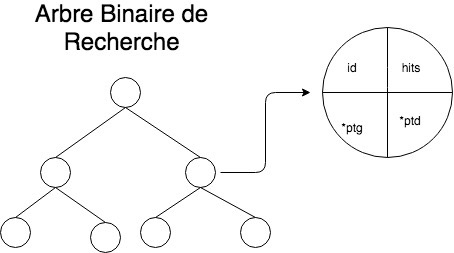
Ceci est l’implementation de notre structure de données. Vu que c’est la classe la plus compliquée, elle sera décrite dans la partie suivante.

IV. Structure des Données

Pour concevoir notre structure de données, nous avons réfléchi d’abord aux opérations qui seraient les plus utilisées. Dans ce cas, nous pouvons dire que c’est l’insertion et la recherche. Ceci nous élimine déjà quelques possibilités comme des listes, listes chaînées, piles et files. Nous avons choisi un Arbre Binaire de Recherche (BST), qui fait les opérations mentionnées en temps quasi-constants ( O(log n) ).

Chaque noeud de notre arbre possède un identifiant de la cible et aussi une deuxième structure des données qui contient des couples <identifiant referer, nombre de consultations>. Pour cette deuxième structure, nous n'avons pas besoin de maintenir l'ordre ou d'accéder à des éléments, simplement de pouvoir ajouter et rechercher très rapidement. La table de Hachage fait cela en temps constant (O(1)) et est donc un choix privilégié.

Nous avons aussi un deuxième BST qui est utilisé pour stocker des pairs contenant l’identifiant de la cible et le nombre de consultations. La premiere structure de données est utilisée surtout pour générer le graphe et la deuxième pour obtenir les n premiers éléments selon le nombre de consultations.



Le Cache des strings :

Vu que nous n'avons pas besoin de maintenir l'ordre ou d'accéder à des éléments, simplement de pouvoir ajouter et rechercher très rapidement, nous avons choisi la Table de Hachage, qui fait cela en temps constant (O(1)).

V. Conclusion

>> We miss Maranzana