Technical Details

胡雨軒

November 26, 2014

2.

Contents

Chapter 1

Méthodes d'accès et de stockage des données

Analyser un fichier et construire une représentation abstraite de l'information qu'il contient à partir des informations que l'on en obtient ligne par ligne est un problème d'analyse syntaxique[1] dont il faut ensuite enregistrer les résultats pour pouvoir les utiliser plus tard. Plusieurs solutions sont envisagées dans le cadre de ce micro-projet.

1.1 Analyse syntaxique d'un fichier texte

Programmation impérative ou tentative de programmation fonctionnelle.

1.2 Stockage des données

L'idée la plus naturelle est d'obtenir en sortie du programme deux fichiers textes qui contiennent une liste de nœuds et une liste d'arêtes mais cette idée mène très rapidement à utiliser une représentation temporaire de la structure de composition des sinogrammes. On peut donc également construire une représentation temporaire des objets, l'afficher puis éteindre le programme sans en garder trace. Il est enfin envisageable d'enregistrer le résultat obtenu dans une base de données pour faciliter son exploitation ultérieure.

[1] Un analyseur syntaxique est un *parser* en anglais.

Nous avons bénéficié des lumières de Messieurs ROBERT et TALEL, professeurs respectivement d'architecture des ordinateurs et de bases de données que nous remercions, pour explorer ces trois voies : représentation temporaire, génération de listes et utilisation de base de données. Ces trois voies posent chacune des questions intéressantes en optimisation et modèle relationnel.

Plusieurs choix s'offrent à nous pour le stockage des sinogrammes. La solution la plus pérenne est de passer par une base de données et donc d'utiliser le disque dur. Cela pose des contraintes de , mais quelle est la solution la plus rapide ? Tous les caractères ont une IDS mais seulement les caractères explicites ont un point de code.

1.2.1 Listes d'arêtes et de nœuds

1.2.2 Base de données MariaDB

Notule Je remercie M. TALEL, qui donne le cours `inf225`, de bien avoir accepté de répondre à mes questions.

1.2.3 La carte de données `PairMap`

Notule Je remercie M. ROBERT, qui donne le cours `inf227`, de bien avoir accepté de répondre à mes questions.

Statégies d'optimisation

La structure de données dans laquelle on stocke les caractères est un dictionnaire qui a pour clef l'IDS d'un caractère (ou son hash). Il faut une structure qui évite complètement les redondances, c'est-à-dire que la forêt des sinogrammes n'ait pas de doublon quel que soit l'ordre dans lequel les caractères sont entrés et quel que soit le détail des IDS. `PairMap` n'a pas vocation a être générique : des structures existent déjà pour cela mais à offrir la solution la plus adaptée au stockage des sinogrammes.

Trois stratégies d'optimisation des accès mémoires :

- Un tas ;
- Un accès aléatoire ;
- Un arbre couvrant.

Stratégie du tas

- Chaque caractère possède ses propres composants ;
- La taille d'un tas est 3 ;
- Solution tributaire des IDS ;
- Explosion combinatoire pour mettre à jour ;
- Analyse de caractère rapide.

Stratégie de l'accès aléatoire

- Lecture et écriture rapide si vraiment aléatoire ;
- Chaque sinogramme contient un tableau de référence de composants.

Stratégie de l'arbre couvrant

- Hypothèse : faiblement connecté ;
- Euh… mais ce n'est vraiment pratique à parcourir !

Un sinogramme comporte trois références vers ses composants et pour l'immense majeure partie deux références. Si le sinogramme est une entrée de la table on a un caractère, un point de code et une IDS. Si c'est un sinogramme induit on n'a qu'une IDS. Certains sinogrammes d'un pan Unicode (*unicode block*) sont décomposés en sinogrammes qui n'ont pas de décomposition dans ce pan mais se décomposent dans un autre pan. L'ordre de décomposition des sinogrammes n'est pas respecté globalement mais seulement à l'intérieur d'un pan. Bien qu'un point de code et une IDS renvoient à une même réalité, il faut les voir comme deux sous-clefs, facette d'une même clef.

Tous les caractères ont une IDS mais seulement les caractères explicites ont un point de code. Les caractères implicites n'ont pas de point de code, seulement une IDS. L'IDS peut donc être vue comme une clef principale à côté de laquelle on place lorsque c'est possible un point de code. L'organisation interne de `PairMap` devra donc être un dictionnaire `(K1, V)` avec `K1` une IDS et `V` une référence vers un objet `Node`. Il y a également un dictionnaire `(K2, K1)` avec `K2` un point de code. Le dictionnaire réciproque `(K1, K2)` n'est nécessaire que pour économiser la résolution

d'une référence : on peut en effet accéder à `K2` très facilement à partir de `K1 : K1 → V.K2`.

L'interface de `PairMap` est finalement un dictionnaire `(K1, K2, V)`.

La classe `TreeMap` peut-elle apporter quelque chose de plus que `HashMap` ?