Dimensionnement structure

Matthieu Delobelle, Anthony Durot

11 juin 2017

Introduction

Ce travail a été réalisé dans le cadre du cours de Programmation Avancée lors du cursus en Informatique, Micro-électronique et Automatique à Polytech-Lille.
La problématique de ce projet était de dimensionner une structure de données ainsi qu'un algorithme optimisés en temps et en mémoire afin d'organiser un fichier de type CSV contenant la liste des oeuvres contenues à la galerie Tate Britain.

Il s'agit d'une collection nationale d'art britannique contenant aujourd'hui plus de 100 000 oeuvres des années 1500 à nos jours située à Londres dans une ancienne prison. Il s'agit de la plus grande collection réservée aux oeuvres d'un même pays du monde.

1 Cahier des charges

L'objectif ici est de stocker et trier les informations concernant les oeuvres possédées par le musée dans un laps de temps minime. Ces informations sont contenues dans un fichier CSV assez volumineux (environ 68000 lignes)

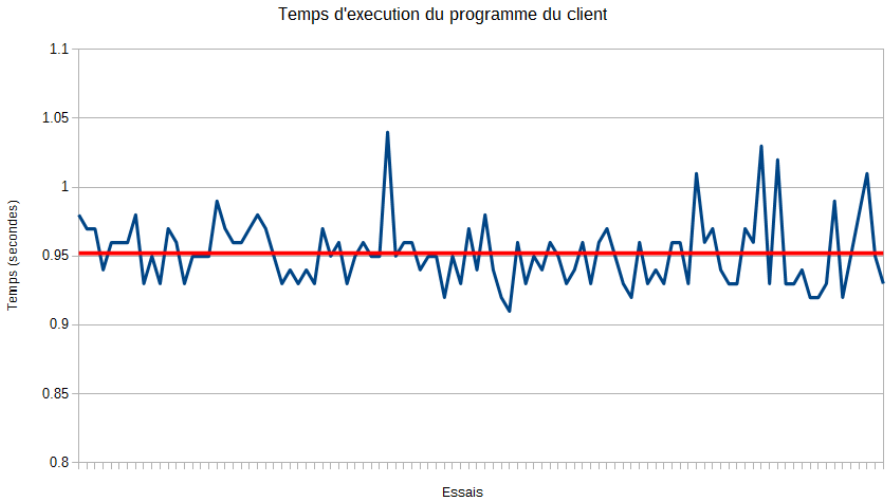
Une fois le fichier lu, il faut stocker les données dans un structure créée spécialement à cet effet. Il faut alors dimensionner cette structure afin de réaliser les opérations le plus rapidement possible.
Ensuite, différentes fonctions de recherche et d'affichage devront être effectuées via une interface utilisateur en console. Dans ce menu, 7 requêtes peuvent être exigées :

- Lister toutes les oeuvres
- Rechercher un artiste spécifique
- Lister toutes les oeuvres d'un artiste spécifique
- Compter le nombre d'oeuvres d'un artiste spécifique
- Afficher le nombre d'oeuvres par artiste
- Afficher l'oeuvre la plus vieille de la collection
- Quitter le menu

L'ensemble de ces requêtes doivent pouvoir être automatisées en chargeant un fichier texte contenant les oeuvres depuis l'entrée standard.

1.1 Exigences supplémentaires

Ce projet est régi par un contrainte de temps d'exécution. En effet, le client (M. Dequidt) dispose déjà, à ce jour, d'une version du programme. Cependant il souhaiterait une version plus rapide.
Voici un graphique reprenant le temps d'exécution moyen du client.



Le programme s'exécute en moyenne en 0.9523 secondes. Nous prendrons ce temps comme contrainte de temps maximum d'exécution. Il faudra donc faire en sorte d'optimiser la structure de données pour profiter d'un temps optimal.

2 Définition de la structure de données

Remarque : les structures de données sont définies dans le fichier **structure.h**

2.1 Informations contenues

Lorsque l'on regarde le fichier CSV on remarque que chaque ligne contient les informations suivantes :

- ID de l'oeuvre dans la base de données
- L'accession number de l'oeuvre. Il s'agit d'un second identifiant permettant de nomenclaturer les oeuvres
- Le nom de l'artiste, son investissement dans l'oeuvre ainsi que son identifiant dans la base de données
- Le titre de l'oeuvre
- Le procédé artistique et le matériau utilisés à la réalisation de l'oeuvre
- L'année de création de l'oeuvre et celle de l'acquisition par le musée
- Les dimensions de l'oeuvre
- Des liens URL menant au site web de Tate Britain au sujet de l'oeuvre.

On définit ainsi une structure Oeuvre contenant toutes ces informations.

2.1.1 Précision du cahier des charges

Lorsque l'on exécute le programme actuel du client, on remarque que certaines informations ne sont pas "importantes" pour ce dernier. On choisit alors, afin de respecter la contrainte de temps d'exécution, de ne traiter que les informations affichées dans la version du client.
On ne traitera donc que les données suivantes :

- L'ID et le titre de l'oeuvre
- L'année de sa réalisation
- L'ID et le nom de l'artiste

2.2 Définition de la structure

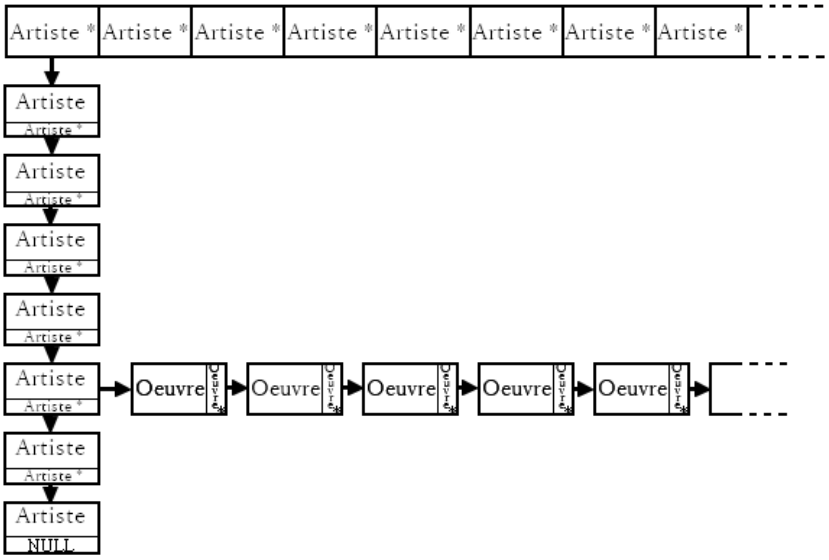
Afin de stocker ces informations le plus rapidement possible, une étude sur les diverses possibilités de structure a été effectuée. Une en particulier nous a paru optimale afin de minimiser les calculs.

Nous partimes donc sur une architecture de ce type :

- Une structure Oeuvre contenant l'ID, le titre, et l'année de réalisation.

— Une structure Artiste contenant l'ID et le nom de l'artiste. Ainsi qu'une liste chaînée contenant toutes les oeuvres réalisées par cet artiste
— Une table de hachage regroupant des listes chaînées d'Artistes.

Voici un schéma reprenant l'architecture utilisée



2.2.1 Taille de la table de hachage

Afin de correctement définir la table de hachage et d'optimiser la rapidité du programme, une étude statistique à été réalisée calculant la moyenne et l'écart-type du nombre de collisions dans la table de hachage en fonction de sa taille. Cette étude à été réalisée grâce à un algorithme en Python que nous avons codé. Vous retrouverez le résultat de cette recherche dans un fichier stat.csv dans la branche stat du projet.

On souhaite une valeur de moyenne de collision de l'ordre de l'unité et un écart-type négligeable. De ce fait, les artistes seraient uniformément rangés en fonction de leur ID dans la table, en évitant toutes cases vides, afin de linéariser et d'optimiser le temps de traitement. On choisira donc une taille de table de hachage de : 334. Pour cette taille, on a une moyenne de 10 collisions par indice, et un écart-type de $3*10^{-4}$.

3 Lecture du fichier CSV et peuplement de la table de Hachage

3.1 Gestion de l'oeuvre la plus vieille

Un pointeur d'Oeuvre étiquette la structure Oeuvre la plus vieille. Ce pointeur est égal à NULL au départ de l'algorithme. Et à la lecture de chaque ligne du CSV, on compare l'année de l'oeuvre actuelle à celle de l'oeuvre pointée par le pointeur, et l'on réaffecte le pointeur en conséquence.

3.2 Hachage

Pour hacher l'ensemble des artistes, on avait repris un algorithme de découpage utilisé dans un précédent travail qui hache en fonction d'une chaîne de caractères, ici le nom. Cependant dans les données, les noms d'artistes ne sont pas uniquement alphabétique. Nous sommes donc partis sur un hachage par ID d'artiste, modulo la taille de la table.

3.3 Parsing du CSV

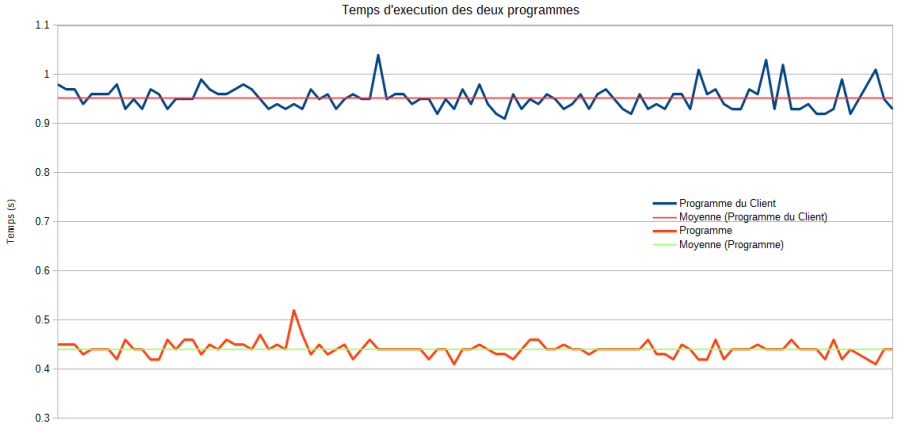
Pour découper le fichier CSV, nous sommes partis de la fonction strtok, qui faisait à priori une chose similaire à nos attentes. Cependant, strtok a des défauts, elle élimine les caractères redondants, et on ne peut pas gérer les quotes avec. Donc nous avons pris la décision de recoder strtok.

La fonction étant open source, nous avons pu nous procurer le code sur internet et le modifier pour qu'il se soumette à nos attentes.
Notre fonction va tout simplement consister en une chaîne que l'on va parcourir. Lorsqu'on observe un "délimiteur" (variable char déclarée plus haut dans le code), on place un '\0' et on renvoie la chaîne (char* info), tandis qu'une chaîne "reste" va maintenant pointer vers la suite. Et on réitère le procédé jusqu'à ce que "info" et "reste" soient NULL, ce qui signifie qu'on a terminé le parsing de la ligne.

Pour gérer les élements de type texte, et donc commençant par des guillemets et contenant potentiellement une virgule. On définit un cas où le premier caractère lu est un ' " ', dans ce cas la variable de délimitation de champ devient également un ' " ' (Fin de la chaîne)

4 Résultats

Avec toute cette architecture et le système mis en oeuvre. On arrive au résultat suivant :



On obtient un temps moyen d'exécution de 0.4409 secondes. Soit environ 2 fois moins de temps que le programme du client. L'objectif est alors atteint.

Remarques : Les essais de temps ont tous été réalisés sur le PC Portable de Matthieu Delobelle (Ubuntu 16.04 64bits, Processeur AMD E1-2100, 4Go de RAM) Vous obtiendrez sûrement des temps d'exécution plus rapides sur vos machines.

4.1 Consommation Mémoire

Le valgrind nous donne ceci, on peut en tirer qu'on a libéré toute la mémoire allouée, et aussi que notre structure représente environ 8 Mo de mémoire. Ce qui est plutôt raisonnable pour des données de cette envergure.

```
==4135== HEAP SUMMARY:
==4135==     in use at exit: 0 bytes in 0 blocks
==4135==   total heap usage: 283,493 allocs, 283,493 frees, 8,480,824 bytes allocated
==4135==
==4135== All heap blocks were freed -- no leaks are possible
==4135==
==4135== For counts of detected and suppressed errors, rerun with: -v
==4135== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Conclusion

L'objectif de ce projet était la mise en place d'une architecture optimisée afin de traiter une demande précise.
Une telle optimisation devient nécessaire lorsque l'on traite de grandes quantités de données. Le fichier ici commence à être d'une taille conséquente, mais n'est rien comparé aux données que peuvent stocker des groupes comme Facebook ou Twitter. On imagine bien l'optimisation nécessaire pour faire fonctionner correctement leurs systèmes.

Des essais pour réaliser une interface graphique ont été faits, mais faute de temps il ne peuvent vous être présentés ici. Vous en retrouverez une esquisse dans le dossier "sdl/" de la branche master.