Rapport: CY Wildwater

v1.0 Projet C-Wildwater

Pré-ing2 MI-2
PINEAU Adam
WALLOT Manfred
KIALA Ronyx

Sommaire

1.Introduction

Ce projet a pour objectif de développer une chaîne de traitement informatique capable d'analyser les données massives du réseau de distribution d'eau en France. L'application combine un script Shell pour l'orchestration et des programmes en langage C pour les calculs intensifs, traitant notamment les histogrammes de capacité des usines et la détection des pertes (fuites) sur le réseau aval.

2. Conception

Notre application repose sur une séparation claire des “responsabilités” : le Shell gère les données et l'interaction utilisateur, tandis que le C prend en charge les structures de données et les calculs.

Pour ce qui est de l’organisation de notre projet, nous sommes organisés de la manière suivante: Manfred était chargé du tris, affichage des histogrammes avec le SHELL. Adam gérait toutes la partie leaks (traçage du parcours aval de l’usine et calcule des fuites ainsi que le partie SHELL associé).Ronyx structuration du C.

2.1Outils de travail:

Pour mener à bien notre projet nous avons choisi d’utiliser la plateforme Discord pour stocker et partager nos progrès étant beaucoup plus familier avec. Github stockait uniquement les fichiers finaux.

Nous avons aussi utilisé l’IA notamment pour corriger certaines erreurs de syntaxes avec le SHELL,elle nous a permis d’installer l'environnement nécessaire pour coder et utiliser efficacement Gnuplot. Etant donné que nous codions sur Windows on a rencontré des problèmes avec les espaces chariots qui empécher le code de se lancer

L’IA a été très utile pour nous fournir des solutions et les commandes appropriées pour que le code puisse fonctionner aussi bien sur Windows que sur Linux. On peut citer la commande” chmod +x “pour autoriser le SHELL à se lancer

Pour le C on a utilisé majoritairement des fonctions vus en cours et en TD.

2.2. Scripts Shell :

Le projet s'articule autour de deux scripts principaux :

- myScript.sh :
  - Il valide les arguments et l'existence des fichiers d'entrée.
  - Il assure la compilation conditionnelle via make si les exécutables sont absents.
  - Filtrage en amont : Pour optimiser les performances, il utilise grep, cut et awk pour ne transmettre au programme C que les colonnes et lignes pertinentes (ex: isolation des lignes d'une usine spécifique pour la commande leaks).
  - Post-traitement : Il récupère les résultats bruts du C pour les trier (sort) et préparer les fichiers destinés à Gnuplot.
- creation_histo.sh :
  - Ce script est appelé automatiquement par le script principal. Il configure et exécute Gnuplot pour générer les fichiers images .png.
  - Il gère la création de deux graphiques distincts (les 10 plus grandes et les 50 plus petites) pour résoudre les problèmes d'échelle mentionnés précédemment.

Figure 1: Exemple de sortie générée automatiquement par le script `creation_histo.sh` (50 plus petites usines par capacité max).

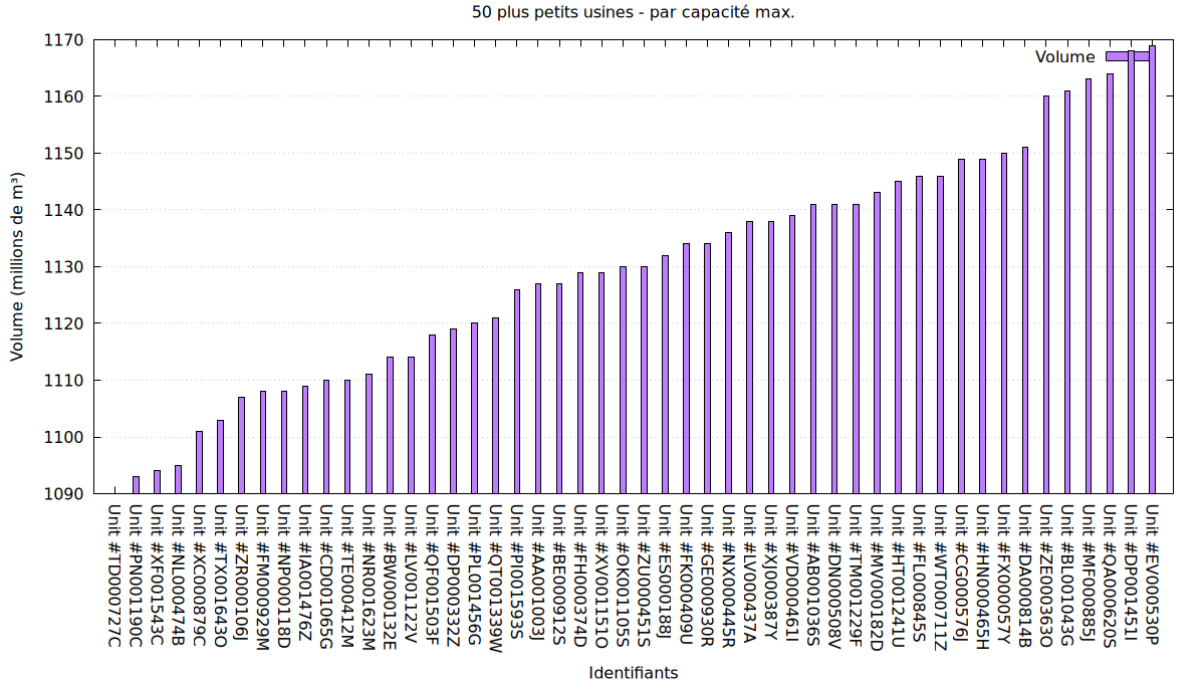
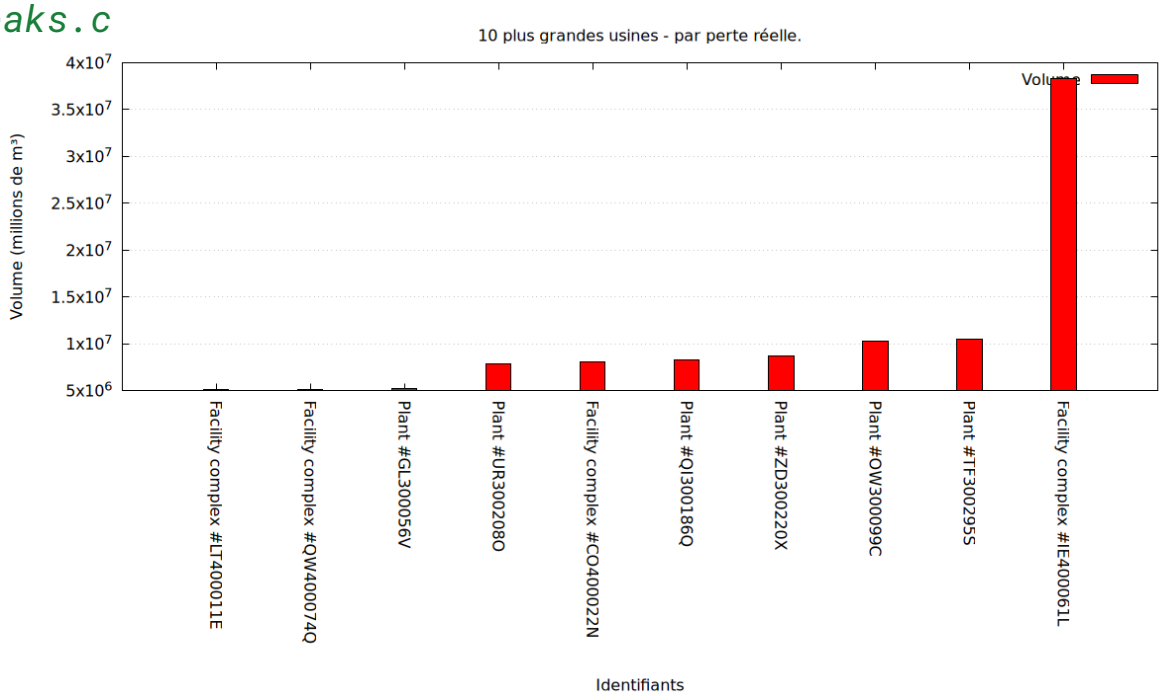


Figure 2: Visualisation des pertes réelles calculées par le programme `leaks.c`



2.3.1. Programme C : histo.c

Remarque : (SUBCOMMANDE désigne le second arguments a mettre dans myScript)

Ce module est dédié à l'assemblage des données pour générer les histogrammes.

Lorsque l'utilisateur saisit un argument **max**, **src** ou **real**, on récupère différentes lignes à l'aide de la commande `grep`, en fonction de la **SUBCOMMAND** choisie par l'utilisateur. Grâce à la commande case et des conditions `if`, des traitements différents sont ensuite appliqués dans le programme en shell et c .

Pour **max**, on récupère simplement l'identifiant(2ème colonne) de l'usine ainsi que sa capacité maximale(3ème colonne) en volume dans les lignes usines.

Pour **src**, on récupère l'identifiant (3e colonne) et les volumes d'eau captés de manière annuelle (colonne 4) dans les lignes *sources-usines* (captage). Pour trier les usines, on utilise la commande `sort`. Pour extraire les colonnes, on utilise ensuite `cut` et `awk`.

Puis, dans le programme en **C**, on crée un AVL dans lequel chaque nœud correspond à une usine. Si une usine ajoutée possède le même identifiant qu'une usine déjà présente dans l'AVL, les volumes sont additionnés.

Pour **real**, on récupère une colonne supplémentaire dans les lignes correspondant au captage : il s'agit du pourcentage de perte. Celui-ci est appliqué au volume de la même ligne avant d'être additionné dans l'AVL, comme pour **src**.

Enfin, ces données sont envoyées au programme **shell** chargé de générer les histogrammes. Celui-ci prend en argument la **SUBCOMMAND** afin d'appliquer la légende et les couleurs appropriées aux graphiques.

Quelques précision sur l'AVL utilisé :

- Structure de données : Utilisation d'un AVL classique (struct avl). Chaque nœud représente une usine et stocke son identifiant, sa capacité, son volume capté et son taux de perte.
- Logique d'insertion (insertAVL) : Contrairement à une insertion standard, si l'usine existe déjà dans l'arbre, le programme met à jour les valeurs (somme cumulée des volumes captés) au lieu de créer un doublon.
- Tri final : Une fois l'arbre rempli, il est converti en tableau via avlToArray puis trié rapidement avec qsort avant l'écriture dans le fichier texte de sortie.

2.3.2. Programme C : leaks.c (Détection de Fuites)

Ce module est le plus complexe algorithmiquement car il doit modéliser la topologie du réseau.

- Structures Hybrides :
  - L'Index (AVL) : Un arbre AVL (struct AVL) sert uniquement d'index pour retrouver l'adresse mémoire d'un nœud en fonction de son ID (O(\log n).
  - Le Graphe (Arbre N-aire) : Les données sont stockées dans des structures Noeud. Chaque nœud possède une liste chaînée dynamique (struct Enfant) pointant vers ses successeurs aval. Cela permet à une usine ou une jonction d'avoir un nombre illimité de connexions sortantes.
- Fonctions Clés :
  - chargerDonnees : Lit le fichier filtré et construit le graphe. Elle gère la création des nœuds à la volée et l'ajout des liens parent-enfant avec leurs ratios de fuite respectifs.
  - calculerTotalPertes : Fonction récursive qui parcourt l'arbre depuis l'usine cible. Elle calcule le volume entrant dans chaque sous-tronçon, applique le pourcentage de perte local, et propage le volume restant aux enfants.
  - Gestion Mémoire (libererReseau) : Une attention particulière a été portée à la libération de la mémoire (nœuds, listes chaînées et index AVL) pour éviter les fuites mémoire.

Exemple d'une ligne du fichier retraçant l'historique des résultats obtenus avec la partie **leaks**.

Usine: Module #SC100266I | Volume Traité: 47.73 Mm3 | Total Fuites: 3.54 Mm3 (Rendement: 92.59%)

3. Difficultés Rencontrées et Solutions

Au cours du développement, notre groupe a fait face à trois défis majeurs qui ont guidé nos choix d'architecture.

3.1Analyse du Sujet

La première difficulté a résidé dans la compréhension globale des attentes et de la structure des données. Le fichier CSV fourni, représentant une topologie complexe (Sources → Usines → Stockages → ... → Usagers), n'est pas dans l'ordre, ce qui a rendu la visualisation des données difficile au premier abord.

La difficulté était donc de savoir comment on allait répartir les tâches qui incombaient au Shell (filtrage) et celles qui incombaient au C (structures de données).

Nous avons réalisé une étude préalable des différents types de patterns décrits dans le sujet (Source-Usine, Usine-Stockage, etc.) pour comprendre que les liens se font via des identifiants uniques. Cela nous a permis de définir les spécifications de nos structures C avant de commencer le code.

3.2. Conception

La fonctionnalité leaks aura été un défi important. La difficulté était double :

1. Modéliser un réseau où chaque nœud (usine, jonction) peut avoir un nombre indéfini d'enfants.
2. Retrouver rapidement les nœuds parents pour construire l'arbre, alors que les lignes du fichier texte sont dans le désordre.

Nous avons opté pour une structure hybride dans le fichier leaks.c.Nous utilisons un AVL qui indexe tous les nœuds par leur identifiant unique. Cela a garanti une complexité de recherche en O(\log n).

    - **Pour la structure du réseau :** Chaque élément de l'AVL pointe vers une structure Noeud contenant une liste chaînée d'enfants (struct Enfant* liste_fils). Cette liste dynamique permet de gérer un nombre illimité de connexions aval sans gaspiller de mémoire.
    - **Algorithme :** Le calcul des fuites se fait ensuite par un parcours récursif de cet arbre n-aire, propageant les volumes du parent vers les enfants via la fonction calculerTotalPertes.

Faute de temps, certaines finitions d'ordre organisationnel n'ont pas pu être implémentées dans la version finale du projet :

- Actuellement, nous utilisons exclusivement des fichiers texte bruts (`.txt` ou `.dat`) pour stocker les résultats intermédiaires et finaux. Bien que cela soit suffisant pour le traitement par le script Shell.
- Nous n'avons pas eu le temps d'ajouter systématiquement des titres pour nommer les colonnes dans les fichiers CSV de sortie. Cela n'empêche pas le fonctionnement du programme ou de Gnuplot, mais aurait amélioré la lisibilité.
- Faute à un soucis d'organisation nous n'avons pas pu uniformiser la langue des commentaires et des noms de variable utilisés
- De plus, l'on a deux main différent, l'un pour la partie leaks et l'autre pour les histogrammes, que l'on aurait aimé rassembler en un seul.

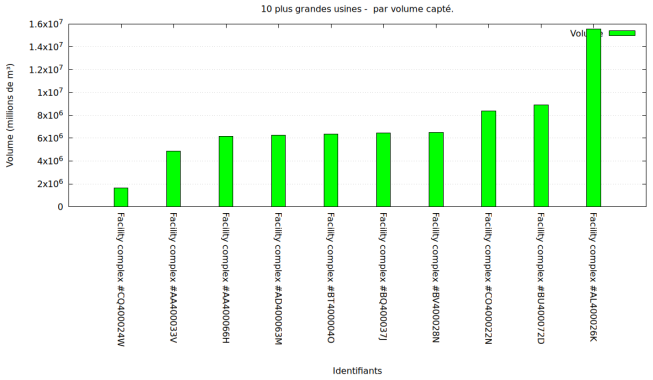
3.3. Visualisation des données

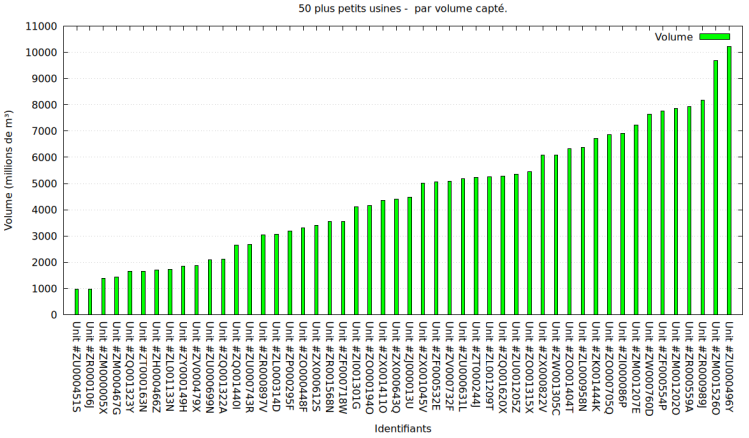
La génération des histogrammes via Gnuplot a posé un problème de lisibilité important. Les écarts de valeurs entre les plus grosses usines (traitant des millions de m^3) et les plus petites étaient tels qu'il était impossible de tout afficher sur un graphique linéaire unique sans rendre les petites valeurs invisibles.

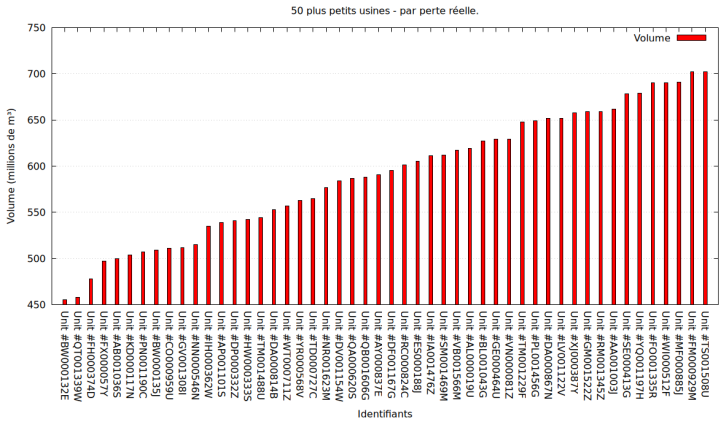
Nous avons décidé de scinder l'analyse dans le script de génération creation_histo.sh. Au lieu d'un graphique unique illisible, nous générons systématiquement deux visuels distincts:

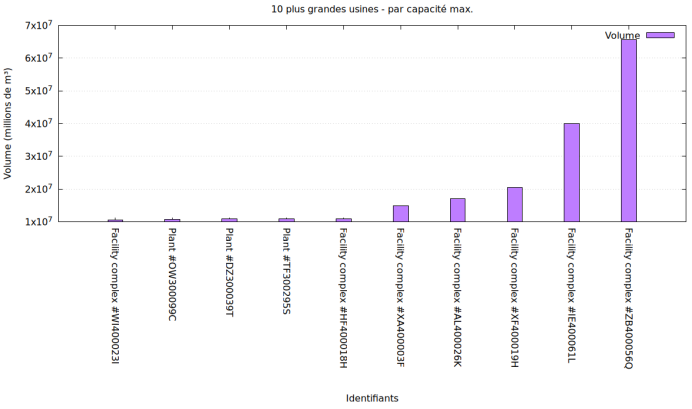
1. Un histogramme des **10 plus grandes usines** (Top 10).
2. Un histogramme des 50 plus petites usines (Bottom 50). Cette approche, couplée à un tri préalable via la commande sort dans le script Shell principal, permet de conserver une échelle pertinente et lisible pour chaque catégorie de données.

Figure 3,4,5,6 :









4. Conclusion

Ce projet nous a permis de mettre en pratique les connaissances acquis durant le semestre entier (AVL, listes chaînées,SHELL) grâce à un cas pratique de traitement de données massives.

Nous avons réussi à surmonter les principaux obstacles, notamment la lecture d'un fichier désordonné et le calcul complexe des fuites, grâce à l'utilisation efficace des AVL. Ces structures nous ont permis de traiter rapidement de grands volumes d'informations sans ralentir le programme.