Rapport de projet PIM

Équipe CD 08 – Paul Louka, Nicolas Bailliet

Janvier 2024

Table des matières

Ce rapport établit une présentation de notre implémentation de l'algorithme de PageRank, sous forme d'un programme exécuté sur l'invite de commandes Linux. Nous détaillerons ici l'architecture générale du programme complet, les principaux algorithmes et structures de données employées, ainsi qu'une explication des décisions menées au cours de la conception. Les dernières parties du rapport illustrent les performances atteintes sur différents graphes d'exemple, et finalement la répartition dans le travail de binôme.

1 Introduction

L'algorithme de PageRank cherche à trier les N différents nœuds d'un graphe selon leur degré sortant, mais aussi le nombre de nœuds qui possèdent un arc l'atteignant, menant à une sorte d'ordre d'importance.

La formule théorique mène à un algorithme itératif pour le calcul des poids de chaque nœud.

$$\pi_{k+1}(n_i) = \sum_{\substack{j \\ (j,i)\in\mathcal{A}}} \frac{\pi_k(n_j)}{|n_j|} \quad \text{pour les nœuds } n_i, \text{ et } \mathcal{A} \text{ l'ensemble des arêtes du graphe}$$

Par souci d'efficacité calculatoire, qui est centrale dans ce projet, on établit une implémentation matricielle équivalente. On définit pour cela la matrice G à partir de S, matrice d'adjacence dont chaque ligne est divisée par le degré sortant, valant $\frac{1}{N}$ sur toute la ligne si le degré est nul. J est la matrice pleine de 1.

$$G = \alpha \cdot S + \frac{(1-\alpha)}{N} \cdot J \quad \alpha \in [0,1]$$

L'algorithme final itératif est le suivant.

$$\pi_0^T = \left(\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N}\right)$$

$$\forall k > 0,\ \pi_{k+1}^T = \pi_k^T \cdot G$$

Nous avons implémenté cet algorithme avec deux méthodes, d'une part avec des matrices classiques dites pleines de complexité spatiale $O(N^2)$, puis d'autre avec des matrices dites creuses, spatialement en $O(|\mathcal{A}|)$.

2 Architecture du programme

Nous avons opté pour une subdivision du programme principal nommé `pagerank` en quelques modules assez simples. Ceux-ci sont les implémentations des matrices pleines dans `Matrix`, et creuses dans `Sparse_Matrix`. Les deux modules ainsi que `pagerank` utilisent dernièrement le module `Vector`, qui a été séparé pour améliorer la lisibilité et éviter les duplications de code. Le schéma suivant illustre l'architecture établie.

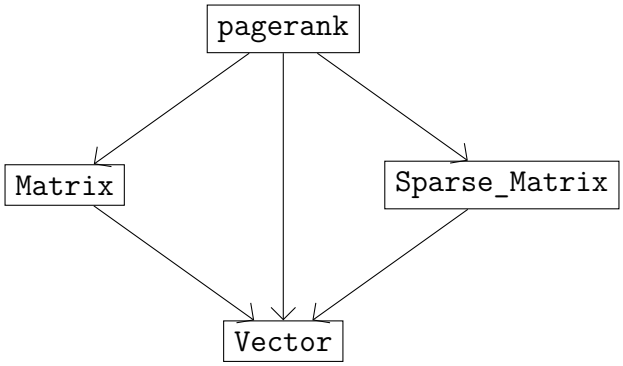

FIGURE 1 – Architecture of the pagerank program.

3 Structures et algorithmes

Le premier choix principal a été d'établir un type universel `T_Element` pour toutes les valeurs manipulées dans les programmes.

```
type T_element is digits <>;
```

Cela permet d'adapter la précision des calculs au besoin. Pour les tests de performance, nous travaillerons avec des `Long_Float`.

La version en matrices pleines de l'algorithme était relativement simple à concevoir, le type `T_Matrix` étant issu d'un type Ada de base.

La version creuse s'avère plus complexe. Nous reviendrons sur les justifications dans la partie suivante, mais notre choix de structure de données pour implémenter les matrices creuses s'est porté sur les Compressed Sparse Columns (CSC). Le type modélise une matrice par trois listes :

- `Values` : la liste des valeurs de la matrice lue par colonne de gauche à droite
- `Row_Indices` : la liste des indices de ligne des valeurs correspondantes
- `Column_Pointers` : la liste de taille $N+1$ des indices délimitant une colonne dans `Values`

```
type T_Elements is array (1..NNZ) of T_Element;
type T_Row_Indices is array (1..NNZ) of Positive;
type T_Column_Indices is array (1..N+1) of Positive;

type T_Sparse_Matrix is record
    Values: T_Elements;
    Row_Indices: T_Row_Indices;
    Column_Pointers: T_Column_Indices;
end record;
```

Pour pouvoir appliquer l'algorithme efficacement, nous utiliserons un vecteur `Coefficients` listant les indices des lignes vides, i.e. des nœuds de degré sortant nul. Ce vecteur est implémenté à l'aide du type générique Ada des vecteurs, `Ada.Containers.Vectors`.

En raison du choix de généraliser le type `T_Element`, mais aussi de la taille du graphe N variable à chaque exécution, l'ensemble des types du programme est rendu générique.

Enfin, une dernière optimisation majeure a été d'employer des algorithmes de tri de complexité temporelle en $O(Nlog(N))$, comme le QuickSort. Par souci de simplicité on utilise les fonctions issues du module générique `Ada.Containers.Generic_Array_Sort`.

4 Conception et difficultés surmontées

La première étape a été la mise en place de la boucle de parsing des paramètres et arguments dans l'appel de l'exécutable `pagerank` depuis l'invite de commande. Nous avons testé un nombre conséquent de possibilités dans les appels et erreurs possibles, pour essayer d'envisager l'ensemble des cas d'arrêt et les traiter avec un message d'erreur personnalisé. Chaque erreur d'utilisation affiche un simple guide d'utilisation avec une description des paramètres et leurs valeurs autorisées.

On a vite remarqué que la valeur N est récupérée dans la lecture des fichiers, i.e. dans le corps du programme principal, or on souhaite instancier les packages génériques correspondant au cas de figure `MODE = Plein` ou `MODE = Creux`. Il nous a donc fallu diviser le programme principal en deux sous programmes internes pour déclarer les variables spécifiques à chaque traitement. Par suite, il a fallu dupliquer une petite partie correspondant à l'enregistrement des résultats du programme.

La version en matrices pleines du programme a été rapide à coder, puisqu'il nous a suffi de suivre l'algorithme matriciel spécifié précédemment. La lecture des valeurs du type `Matrix` est immédiate par le double indiçage, ce qui rendait cela possible. Cependant ce n'est pas le cas pour les matrices creuses.

Pour choisir notre implémentation des matrices creuses, une observation a été cruciale : dans l'algorithme de PageRank, les seules valeurs vraiment «intéressantes» sont celles qui correspondent à la présence d'un arc dans la matrice d'adjacence. Bien que la matrice G est strictement positive, les valeurs qui à l'origine étaient nulles s'infèrent à partir du vecteur `Coefficients` qu'on a créé avec cet objectif en tête. En bref, l'idée générale était d'adapter l'algorithme matriciel pour tirer avantage le plus possible de la faible densité des arcs du graphe. Nous sommes parvenus à formaliser la formule itérative utilisée avec l'instance de `Sparse_Matrix`.

$$\pi_{k+1}(n_i) = \sum_{\substack{j \\ (j,i)\in\mathcal{A}}} \alpha \cdot \frac{\pi_k(n_j)}{|n_j|} + \sum_l \alpha \cdot \frac{\pi_k(\texttt{Coefficients}(l))}{N} + \frac{1-\alpha}{N}$$

Avant de pouvoir utiliser cet algorithme, il était essentiel de pouvoir intialiser la structure CSC efficacement. Hors, par définition du type, cela nécessite d'avoir les arcs triés par colonne en premier. Notre méthode est d'effectuer une première passe pour lire tous les arcs dans le fichier spécifié, les stocker, puis les trier avant de les transformer en CSC.

5 Analyse des performances

Graphe	`Plein`	`Creux`
sujet	0,003	0,007
worm	0,084	0,008
brainlinks	N/A	3,553
linux26	N/A	4,363

TABLE 1 – Performances temporelles (s) sur dragon.enseeiht.fr

6 Conclusion

Nous avons accompli tous nos objectifs avec des temps vraiment satisfaisants. Si on le souhaite, on peut implémenter notre propre version de QuickSort pour les trics des `Arcs` et du vecteur `PI` final, au lieu d'importer le module `Ada.Containers.Generic_Array_Sort`. Si l'implémentation est correctement optimisée, on devrait avoir des performances similaires.

7 Annexe

Apports personnels Nous avons beaucoup aimé ce projet, notamment l'utilisation de `gprof` pour mesurer l'efficacité de chaque fonction dans le programme, ce qui nous a permis de pousser l'optimisation temporelle à ce point. Les algorithmes n'étaient pas les plus complexes, mais savoir tirer parti de sa machine et éviter les calculs redondants est parfois la partie la plus fastidieuse.