INP TOULOUSE ENSEEIHT

Rapport de projet de programmation impérative

Implémentation d'un algorithme de pageRank

Maxime Dubaux
Laurent Fainsin

Département Sciences du Numérique - Première année
2020 - 2021

Table des matières

1 Introduction

Au cours de ce projet, nous avons implémenté un algorithme permettant de calculer le PageRank pour un réseau donné. Nous l'avons construit à partir de la méthode des raffinages. De plus, nous utilisons le langage de programmation compilé Ada.

2 Résumé du sujet

Le PageRank est un algorithme d'analyse des liens entre des pages Web dans un réseau, il mesure la popularité des pages dans celui-ci. En outre, il trie les pages internet de la plus populaire à la moins populaire selon des principes simples :

— Une page est respectable si d'autres pages référencent, et d'autant plus si ces dernières elles même sont populaires.
— Néanmoins, il faut ajuster la popularité selon la page qui référence, plus une page possède de lien vers d'autres sites, moins ses référencements ont de valeur.

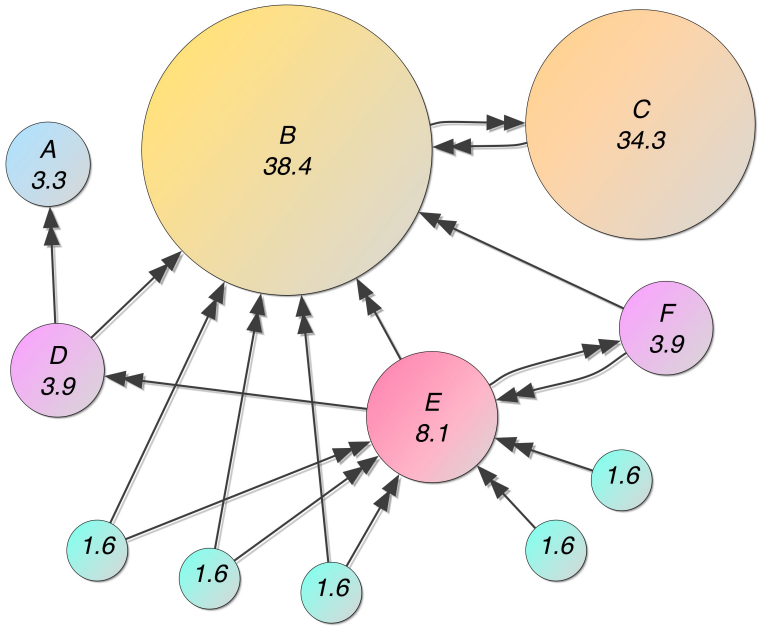

FIGURE 1 – Exemple d'un réseau et de ses poids calculés via le pageRank [1].

À l'aide du théorème du point fixe, il peut se résumer en un calcul répété d'un produit vecteur-matrice.

Toute la difficulté provient de la taille massive que peut avoir le réseau. Il faut donc choisir les bonnes structures de données. Cet algorithme est notamment utilisé par le moteur de recherche Google qui estime son propre réseau à une cinquantaine de milliards de pages.

3 Architecture du programme

Nous pouvons diviser notre programme en trois sous-parties qui s'occuperont de gérer les différents types ainsi que la bonne exécution du programme. De même, mise à part pour la récupération des arguments dans la ligne de commande, nous adopterons un style de programmation offensif puisque l'utilisateur n'interagit jamais avec le programme.

Il est aussi à noter que nous avons choisi d'utiliser des "declare" plutôt que des sous-procédures auxquelles on ferait appel, puisque cela permet d'avoir une lecture plus linéaire du code.

3.1 Gestion des vecteurs (vector.ads)

Ada est un langage fortement typé. Comme nous manipulons ici plusieurs types de données, il est logique de créer un type "Vector" générique dont l'unique but sera de stocker des informations. Nous aurons aussi besoin à plusieurs moments lors du calcul du pagerank de trier, sommer, initialiser des Vectors, d'où le choix de placer cette structure de données dans son propre module.

Lorsque nous trierons les données, nous ferons appel à l'algorithme QuickSort.

Nous avons séparé le module Vector en trois sous-modules : Un module capable de stocker des entiers, un autre pour des flottants et un dernier pour des liens. Nous avons fait ce choix car créer un unique module générique Vector pour gérer ces trois types de données (très différents) était trop compliqué. Cela était tout de même faisable mais le code était compliqué à lire. Ainsi bien que le code soit quelque fois redondant, il est plus compréhensible.

3.2 Gestion des matrice de Google (google_*.ads)

Nous devrons regrouper dans des modules génériques le code en rapport avec la gestion des matrices de Google. Nous utiliserons deux modules, un gérant les matrices naïves et un autre pour gérer les matrices creuses.

Ces modules introduiront le type T_Google, ainsi que des procédures et fonctions permettant de générer la matrice G, nécessaire au calcul du pageRank. Ils implémenteront aussi l'opération de multiplication entre un vecteur et une matrice.

3.3 Gestion du calcul du Pagerank (pagerank.adb)

Cette dernière partie s'occupe de regrouper tous les éléments présents dans les deux modules cités précédemment pour ainsi calculer itérativement le pageRank du réseau. Cette sous-partie gère de plus le traitement des arguments de la ligne de commande ainsi que la lecture et l'écriture des résultats dans des fichiers.

4 Structures de données

Nous avons besoin d'une structure pour stocker le poid de chaque pages (i.e. sa popularité), nous appellerons cette structure "pi". De même, nous avons besoin d'une structure pour stocker le réseau "network" décrivant les liens entre chaque page du réseau. Enfin il nous faudra créer une structure de données adaptée à la taille du réseau pour stocker la matrice de Google, notée G.

Nous connaissons à l'avance les tailles des différentes structures, car le réseau est connu. Il n'est donc pas nécessaire de créer des types dynamiques (i.e. liste chainées, vecteurs à taille variable stockés dans le heap...), un simple stockage statique dans le stack suffit.

Notons N le nombre de pages dans le réseau, et N_Links le nombre de liens total dans le réseau.

Ainsi pour stocker pi, nous choisissons la structure d'un vecteur de flottants de dimension 1xN.

De même, pour stocker network nous choisissons la structure d'un vecteur de T_Links de dimension 1xN_Links (avec T_Links un enregistrement permettant de stocker les informations d'un lien).

Pour stocker G, la matrice de Google, nous avons 2 choix :
— Une Matrice naïve
— Une Matrice creuse (éventuellement compressée)

4.1 Implémentation Naïve

Cette première implémentation consiste à stocker la matrice très naïvement, c'est-à-dire en stockant l'ensemble de ses valeurs dans une matrice de dimension NxN.

L'avantage principale de cette structure est que sa construction et sa manipulation (produit vecteur-matrice) est simple. Pour la construire à partir du réseau nous assignons à chaque lien une valeur dans la matrice. De même, celle-ci a l'avantage par construction d'être robuste par défaut à la présence de doublons dans le réseau.

L'inconvénient est que nous stockons beaucoup de valeurs inutiles (i.e. des éléments qui se répètent ou bien qui sont égaux à 0). Ainsi, puisqu'un produit vecteur-matrice est de complexité N^2, nous perdons beaucoup de temps à effectuer des opérations inutiles

4.2 Implémentation Creuse

Il faut alors nous orienter vers une autre solution si l'on souhaite être plus efficace.

Pour alléger la taille mémoire de la matrice G, nous pouvons la rendre creuse, c'est-à-dire ne pas stocker les valeurs nulles de celle-ci. Cependant nous pouvons aller encore plus loin en compressant G via un algorithme de compression simple appelé CSR [2].

De cette manière nous ne gardons que les informations qui sont essentielles.

L'inconvénient de cette méthode est que la création de ce type de matrice et son utilisation sont plus compliquées que pour une matrice naïve.

Mais l'avantage de cette structure de données est son gain d'espace non négligeable ainsi que la rapidité qu'elle propose. En effet, nous n'effectuons plus que N_Links opérations lors du produit vecteur-matrice, au lieu de N^2 pour la version naïve. De même, nous pouvons aussi nous permettre de stocker uniquement des entiers dans G, ce qui diminue encore plus la complexité spatiale.

5 Benchmark

Voici l'ensemble des tests réalisés avec la commande time et valgrind sur l'ordinateur c202-02 de l'ENSEEIHT.

La commande suivante a été éxécutée pour élargir la taille maximale du stack :

```
$ ulimit -s unlimited
```

Valgrind génère une sortie similaire à celle-ci pour l'ensemble des programmes :

```
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 22 allocs, 22 frees, 27,308 bytes allocated

All heap blocks were freed -- no leaks are possible

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Comme le heap n'est pas utilisé par le programme (mis à part pour quelques variables dont nous n'avons pas le contrôle direct), sachant le PID du programme, nous pouvons naïvement inspecter le fichier suivant pour connaître la taille qu'occupe le programme dans le stack :

```
$ build/pagerank fichiers_test/Exemple_sujet/exemple_sujet.net -n & \
    { while [[ -f /proc/$!/smaps ]]; do grep -A 1 stack /proc/$!/smaps >> stack.txt; done } ; \
tail stack.txt
```

5.1 Version Naïve

5.1.1 exemple_sujet.net

Stack size : 132 kB

```
$ time build/pagerank fichiers_test/Exemple_sujet/exemple_sujet.net -n
real        0m0,009s
user        0m0,000s
sys         0m0,006s
```

5.1.2 worm.net

Stack size : 1252 kB

```
$ time build/pagerank fichiers_test/Worm/worm.net -n
real        0m0,073s
user        0m0,061s
sys         0m0,009s
```

5.1.3 brainlinks.net

Stack size : 1570492 kB

```
$ time build/pagerank fichiers_test/Brainlinks/brainlinks.net -n
real        2m41,387s
user        2m41,074s
sys         0m0,280s
```

5.1.4 Linux26.net

Le programme utilise trop de place dans le stack, il ne peut donc pas être exécuté. On peut tout de même estimer son espace mémoire à au moins 650 Go.

5.2 Version Creuse

5.2.1 exemple_sujet.net

Stack size : 132 kB

```
$ time build/pagerank fichiers_test/Exemple_sujet/exemple_sujet.net
real        0m0,168s
user        0m0,001s
sys         0m0,009s
```

5.2.2 worm.net

Stack size : 132 kB

```
$ time build/pagerank fichiers_test/Worm/worm.net
real        0m0,034s
user        0m0,016s
sys         0m0,004s
```

5.2.3 brainlinks.net

Stack size : 14748 kB

```
$ time build/pagerank fichiers_test/Brainlinks/brainlinks.net
real        1m55,939s
user        1m55,909s
sys         0m0,012s
```

5.2.4 Linux26.net

Stack size : 41152 kB

```
$ time build/pagerank fichiers_test/Linux26/Linux26.net
real        437m38,234s
user        437m34,783s
sys         0m0,440s
```

6 Conclusion

Nos programmes ne génèrent aucune erreur selon Valgrind et s'exécutent pour chacun des réseaux (mis à part pour Linux26 dans le cas naïf, mais cela semble normal au vu de la taille du réseau).

Les fichiers .org et .p qu'ils génèrent sont aussi pratiquement identiques à ceux fournis par l'énoncé. Il y a parfois quelques différences dans les fichiers .ord car certaines pages ont la même valeur de poids et parce que nous n'avons probablement pas utilisé le même algorithme de tri. De même les seules différences dans les fichiers .p sont dans les décimales après la précision donnée.

On remarque facilement la supériorité temporelle et spatiale de la version creuse contre la version naïve, surtout lorsque N et N_links sont grands.

6.1 Améliorations encore possible

Lors de l'implémentation de la matrice compressée, nous avons choisi de compresser les lignes puisque celle-cis peuvent parfois être entièrement vides, cela est bénéfique d'un point de vue l'espace. Cependant il serait aussi intéressant de compresser G selon les colonnes puisque, bien que l'on perde en espace mémoire, on gagnerait en efficacité temporelle grâce au nombre réduit d'accès mémoire que l'on effectuerait par rapport à la compression par ligne.

Il est important de noter qu'une compression par colonne permettrait aussi de paralléliser le problème, le rendant encore plus efficace temporellement. Il est assez simple d'implémenter la parallélisation en Ada puisque cette notion fait partie à part entière du langage de programmation, mais nous ne l'avons pas implémenté par manque de temps.

Il serait aussi possible de rendre l'algorithme QuickSort générique mais nous ne l'avons pas non plus fait par manque de temps.

Finalement une dernière amélioration possible est celle de supprimer network du scope lorsque nous n'en n'avons plus besoin. En effet , une fois le réseau créé à partir du fichier .net, nous le gardons jusqu'à la fin de l'exécution du programme. Cependant il est possible de s'en débarrasser après la création de G.

7 Apports personnels

Ce projet nous a permis de consolider nos connaissances sur les structures de données, car nous avons longuement réfléchi lesquelles étaient les plus adaptées au problème. De même, nous avons eu l'occasion de revoir plusieurs algorithmes classiques, nous permettant de mieux les maîtriser. Nous sommes tout de même déçus de ne pas avoir eu plus de temps à consacrer à l'optimisation du programme.

Références

[1] Wikipedia, PageRank
https://en.wikipedia.org/wiki/PageRank

[2] Wikipedia, Sparse matrix
https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)

[3] Wikibooks, Ada Programming
https://en.wikibooks.org/wiki/Ada_Programming

[4] Rosetta Code, Ada
https://rosettacode.org/wiki/Category:Ada