Rapport de projet PIM

Hammi Kamal et Basile Gros

Plan du rapport

Résumé

Dans ce rapport, nous présentons notre projet de développement d’un algorithme PageRank en Ada. Nous rappelons tout d’abord ce que le projet nous demandait, puis nous expliquons l’architecture de nos programmes. Nous Détaillons ensuite comment nous les avons lis au point et quels problèmes nous avons rencontrés. Enfin nous mettons notre code en perspective avec ce que l’utilisateur, ici Google, demande de l’algorithme PageRank.

Introduction du problème

L’objectif de ce projet était de mettre au point deux implantations de l’algorithme PageRank qui permet à Google de trier dans l’ordre décroissant de popularité des pages Web en fonction des hyperliens entre elles. L’une de ces implantations demandait à ce que la matrice de ces liens soit codées naïvement en tableau carré de côté le nombre de pages, tandis que l’autre demandait une matrice creuse, où seules les valeurs non nulles sont représentées. La première prend une grande place mémoire mais est rapide à exécuter, tandis que dans la deuxième, l’optimisation en temps a été un problème majeur auquel nous nous sommes confrontés.

Rappel du sujet

Pour calculer quelles sont les pages les plus populaires, PageRank consiste à assigner à chacune d’entre elles un poids flottant compris entre 0 et 1 qui correspondra à la popularité de la page, le poid le plus fort étant la page la plus populaire.

Pour calculer ce poids, on calcule une matrice de liens entre les différentes pages. Si une page a un lien vers deux autres pages, les éléments correspondant dans la matrice auront un poids de ½. Si une page ne pointe vers personne, le poids de sa ligne sera de 1/N avec N le nombre de pages. Ce assure que la somme des éléments de chaque ligne soit égale à 1, et empêche une divergence vers l'infini ou zéro des poids. De plus, cette pondération donne de la valeur aux pages ayant peu de lien, évitant la trop grande influence des pages servant d'index.

Cette matrice S est ensuite pondérée en une matrice G dite matrice de Google par un coefficient alpha (α) compris entre 0 et 1 :

$$G = \alpha \cdot S + (1 - \alpha)/N * E$$

avec E une matrice de la taille de S remplie de 1

Ce réel alpha est appelé le *dumping factor*. Plus il est proche de 1 et meilleur sera le résultat mais plus lente sera la convergence.

Le vecteur des poids se calcule alors par la formule : $\pi k+1T = \pi k\ T \cdot G$ à l'itération k
On initialise tous ses éléments à la valeur 1/N pour assurer que la somme des poids de toutes les pages sera égale à 1.
Par défaut, nous prendrons $\alpha = 0.85$ pour assurer un bon compromis entre la rapidité et la précision des calculs. Nous prendrons aussi 150 itérations sauf contre exemple de l'utilisateur.

La ligne de commande à détecter est :

./pagerank -P -I 150 -A 0.90 exemple_sujet.net

Les paramètres -P, -I et -A sont optionnels. L'option -I permet de spécifier le nombre maximal d'itérations et l'option -A de modifier la valeur d'alpha. Le paramètre -P permet d'utiliser l'implantation naïve. Par défaut, on lance l'implantation creuse.
Le fichier .net est le fichier où les liens entre les pages sont indiqués. Sur sa première ligne il y a le nombre total de pages, et chaque ligne suivante aura l'indice de la page du lien suivi de l'indice de la page vers laquelle le lien mène.

Deux fichiers sont générés :
un fichier .p dont la première ligne comporte le nombre de pages, la valeur de alpha et le nombre d'itérations effectuées. Les lignes suivantes ont chacune un poids écrit, par ordre décroissant de popularité des pages correspondantes.
un fichier .ord dont chaque ligne comporte l'identifiant d'une page par poids décroissant.

Architecture des applications

Le programme principal est composé seulement d'une partie qui lit les différentes commandes de l'utilisateur et les interprète. Il charge ensuite le module de l'implantation souhaitée, naïve ou creuse, et gère les erreurs liées à une mauvaise entrée des paramètres (la robustesse).

L’implantation naïve du programme commence par calculer la matrice S en fonction des données lues dans le fichier d’entrée. Une fois cela fait, elle itère le nombre de fois voulu le calcul des poids en y calculant au passage le coefficient de la matrice G. Elle génère ensuite un vecteur des indices des pages, puis trie les pages en fonction de leur poids. Ce tri (Quicksort) est effectué en fonction des poids des pages mais est aussi répercuté sur le vecteur des indices, pour que l’indice d’une page et son poids aient la même place dans leurs matrices respectives.
Enfin, ces données sont écrites dans les fichiers correspondants.

L’implantation creuse possède une architecture similaire. La matrice est codée comme un tableau à une dimension des vecteurs d’adjacences correspondant aux colonnes de la matrice des liens.
Le parcours de la matrice est modifié car un parcours par ligne comme dans la première matrice a une complexité en O(N3). Le nouveau parcours a alors une complexité en O(N2).

Principaux algorithmes

L’algorithme permettant de comprendre la ligne de commande est une imbrication de structure de contrôle if..then..else portant sur les valeurs des différents paramètres. Il donne aux variables correspondantes les valeurs entrées ou les valeurs par défaut et charge alors les modules génériques avec ces valeurs.

Dans l’implantation naïve, le calcul de la matrice S consiste à parcourir le fichier ligne par ligne avec une boucle for et à enregistrer les liens entre les éléments avec le coefficient 1.0 dans la matrice.
Après cela, on parcourt cette matrice par ligne, et en fonction du nombre d'éléments dans chaque ligne, on en change les coefficients.
Le produit matriciel entre la matrice G et les poids est une double boucle for avec une somme du produit des coefficients de la matrice par les poids de l’itération précédente.

Dans l’implantation creuse, en plus de remplir la matrice S par les 1.0, on incrémente l'élément correspondant dans un vecteur stockant les tailles des lignes. Cela évite un parcours par ligne qui, dans notre implémentation, a une complexité temporelle en O(N3).
Le calcul de la matrice H est un parcours de chaque colonne par un curseur dans une boucle for. Si la valeur de l’indice du curseur correspond à celle de l’itérateur, on change l’élément grâce au vecteur de taille des lignes et on passe à la case suivante de la colonne. Sinon, comme les cases de la colonne sont toujours indicées dans l’ordre croissant, on sait qu’on a dépassé l’itérateur et on ajoute une case avec l'itérateur comme indice avant la case pointée par le curseur.

L’algorithme de tri est un algorithme Quicksort décroissant que nous avons codé en nous inspirant de plusieurs implantations différentes de la page Wikipedia.

Explication de la mise au point des algorithmes

Après un raffinage en trois couches pour nous donner une bonne idée de l'architecture que notre programme prendrait, nous avons développé deux modules génériques, un pour chaque implantation.
Nous avions pour idée de donner la même interface à chaque module pour écrire le programme dans le programme principal et n'avoir qu'à changer l'appel de module en fonction de ce qui était demandé.

Malheureusement, nous avions besoin de rendre nos modules génériques à cause notamment de la taille des matrices variant en fonction du nombre de pages à classer. Il nous fallait donc instancier des modules après avoir traité la ligne de commande, et donc au milieu du programme.
Après des recherches sur le site de AdaCore et confirmation auprès de notre professeur, nous avons utilisé la structure **declare** qui permet en Ada de déclarer et d'instancier au milieu d'un programme.
Néanmoins, comme il n'est pas possible d'écrire des structures de contrôle dans la zone de déclaration, cela signifiait que nous allions devoir écrire le programme en double. Quitte à faire cela, nous avons écrit les deux programmes dans leur module respectif.

L'algorithme de tri choisi est le Quicksort puisque les vecteurs de poids et indices ne seront a priori pas ordonnés, et le Quicksort est rapide à les ordonner en ordre décroissant dans ce cas. C'est aussi un algorithme simple à coder en Ada et qui ne prend pas beaucoup de place mémoire.

Dans la version naïve, nous avons privilégié de parcourir la matrice plus souvent si cela nous permettait de gagner en déclaration de vecteurs, car la mémoire est le problème principal de cette implantation, et nous avons préféré le mitiger au détriment du temps d'exécution.

A notre premier essai d'exécution, nos résultats étaient faux, nous avions oublié d'itérer l'algorithme autant de fois que demandé.
A notre essai suivant, les résultats étaient toujours faux, et notre professeur nous a signalé que notre vecteur des poids était modifié pendant le produit matriciel, affectant les calculs après la première valeur.
Nous avons corrigé cela en créant une fonction séparée qui s'occupait du calcul en question.

Alors, nos que deux modules fonctionnaient, sur le fichier le plus simple, ce n'était pas le cas pour l'exemple worm.net. Après avoir cherché dans le programme ce qui n'allait pas, on se rend compte que le fichier .ord de correction pour worm n'était pas bon. Après l'avoir téléchargé à nouveau, les résultats se correspondent bien.

Puisque nous avions codé nos matrices creuse comme un tableau de vecteurs d'adjacence représentant chacun une colonne, le parcours en ligne nécessaire pour calculer les coefficients de S avait une complexité en O(n3) car il fallait à chaque changement de colonne parcourir la suite de pointeur permettant de revenir à la ligne où nous étions. Nous avons donc décidé de créer le vecteur qui enregistre le nombre d'éléments par ligne au moment où les indices sont lus dans le fichier .net.

Avant de faire cela, notre temps d'exécution pour le fichier brainlinks.net était de cinq heures. Il est descendu à une minute après-coup.
Notre dernière erreur est venu de la façon dont on comptait le nombre d’éléments par ligne. Chaque fois que nous avions une nouvelle connexion, nous ajoutions 1 à la case de la ligne, sans vérifier que cette connexion n'existait pas au préalable.
Nous avons donc créé le sous-programme Affecter_Comptage qui vérifie si la case est vide ou non avant de la remplir est d’incrémenter le compteur du nombre d'éléments sur la ligne.

Durées d'exécution moyenne sur les machines de l’école via X2go (Alpha = 0.85, 150 itérations)

exemple_sujet.net :

Implantation creuse : 0.00 secondes
Implantation naive : 0.00 secondes

worm.net :

Implantation creuse : 0.04 secondes
Implantation naive : 0.04 secondes

brainlinks.net :

Implantation creuse : 56.17 secondes
Implantation naive : 170.64 secondes Mémoire allouée : 48 Mo

Conclusion sur l’avancement du projet et les perspectives d’améliorations/évolution

Même avec les différentes étapes d’optimisation de l’implantation creuse par lesquelles nous sommes passées, il est toujours possible de raffiner encore plus cette partie. Si nos données avaient été plus nombreuses, on aurait pu créer un matrice avec une double adjacence, verticale et horizontale.
Dans la perspective de Google, qui possède d'immenses serveurs de données à disposition, et pour qui le temps est la dimension à optimiser en priorité, une implantation qui optimise le temps d'exécution en dépit de l’espace mémoire serait à prioriser.

Apports personnels

Basile Gros

Ce projet a été mon premier vrai travail de programmation en groupe. Il m’a appris qu’il vaut mieux travailler moins souvent mais ensemble et se coordonner sur ce qu’il y a à faire que de partir chacun faire une partie dans son coin.
Il m’a aussi appris à exprimer clairement mon raisonnement et ce que je veux faire, ou pourquoi je pense qu’une idée n’est pas bonne.
Enfin, ce projet m’a fait vraiment découvrir l’importance des coûts en temps et en mémoire, et de l’équilibre à maintenir entre les deux à un niveaux autre que purement mathématique.

Kamal Hammi

Durant mon travail sur ce projet, Je me suis sorti de mon propre espace clos pour confronter celui des autres afin de programmer ensemble impérativement. J'ai bien discuté de tout dès le début, et je ne prends pas en compte une idée que si je suis totalement convaincu par les bons raisonnement exprimés. Le travail en groupe m’a corrigé des fausses compréhension de quelques concepts que j’ai pu avoir, et j’ai appris comment faire pour être prudent en respectant l’heure de rendez-vous fixée au préalable.