ZAFIZARA-BENETRIX Nathan
WATRIN Claire
L2 MIDO

# Processus de réalisation du projet

Légende : — — — fonctions abandonnées ou n'existant plus

## Etape 1 : Implémentation de la structure GRAPHE

Concernant cette étape, primordiale pour pouvoir réaliser nos calculs, nous avons longuement débattu sur la meilleure façon de procéder. Nous avons conservé la structure LinkedList (utilisant une structure auxiliaire Node) telle que nous avons eu l'habitude de l'implémenter en TD/TP. Nous avions pensé à créer une structure graphe mais il nous a finalement paru plus simple de seulement considérer ceux-ci comme des tableaux de listes chainées, i.e de type LinkedList **.
Nous avons ajouté des fonctions « de base » pour traiter la structure LinkedList (createNode, createLinkedList) et nous avons également ajouté des fonctions plus spécifiques pour répondre à nos besoins :

- Une fonction duplicateCheck pour respecter la consigne « *Plusieurs liens vers la même page dans une page donnée ne compte que pour un seul arc* » (partie 4.4 de l'énoncé). Concernant cette fonction, nous nous sommes rendu compte qu'elle était quasiment identique à whatPage et predecessor, nous l'avons donc supprimé.
- Une fonction superNoeud qui retourne un pointeur sur le supernoeud (celui ci pointe sur toutes les pages).
- Une fonction createGraph renvoyant le graphe obtenu : pour tout i=1,…,nbPages+1 G[i] est un pointeur sur le 1er élément d'une liste chainée : Adj[i]->first est la page à laquelle on s'intéresse, le reste de la liste chainée sont ses liens sortants.
- Une fonction whatPage qui renvoie l'indice dans le graphe d'une chaine de caractère passée en argument (ou -1 ci celle-ci n'y apparaît pas).
- Une fonction predecessor qui retourne 1 si la page se trouvant à l'indice i (dans le graphe) est un prédécesseur de la page dont le nom est s, la chaine de caractère passée en argument.
- Une fonction redLinks qui supprime tous les liens rouges du graphe. Cette fonction est plutôt complexe et prend beaucoup de temps. Nous avons eu deux versions de cette fonction. La première parcourait tous les Adj[i]->first->name et vérifiait l'existence grâce à eux. Cette version mettait plus de 40 secondes sur wiki-venetian. La nouvelle version parcourt tout le supernoeud car, par définition, si une page existe alors elle est pointée par le supernoeud. Celle ci met environ 13 secondes sur le même fichier, soit un gain de 27 secondes.

Remarque : la fonction createLinkedList n'est pas une fonction créant une simple liste chainée. En effet, elle nous permet de rendre le graphe *fortement connexe (*avec l'aide de la fonction superNoeud) ; c'est pourquoi, à la fin de la liste chainée G[i], on rajoute la page elle même puis le supernoeud (toutes les pages pointent vers le supernoeud et vers elles mêmes).

Lors de cette étape, nous avons rencontré un problème : tous les Adj[i] représentaient la même liste chainée (la dernière du fichier wiki). Nous avons résolu ce problème grâce au message Teams d'Emmanuel Lazard sur la gestion des chaines.

## Etape 2 : Exploitation des fichiers tests

Pour simplifier l'accès aux données, nous avons crée un tableaux de str (appelé « file_lines ») tel que chaque case du tableau corresponde à une ligne du fichier. La fonction getLines, qui initialise

ce tableau, renvoie le nombre de lignes du fichier (i.e le nombre de pages web), donnée essentielle à notre programme puisqu'elle nous renseigne notamment sur le nombre de sommets de notre graphe (auquel il faut ajouter 1 pour prendre en compte le supernoeud) et est régulièrement utilisé dans les calculs du PageRank.

**Etape 3 : Calcul des probabilités**

Nous avons tout d'abord pensé le calcul des probabilités par rapport aux listes adjacentes (graphes), puis nous avons adapté notre algorithme aux matrices d'adjacence (cf étape 4).
Dans un premier temps, grâce à la fonction initProbas, nous avons initialisé les probabilités de chaque page tel qu'il l'était indiqué dans le sujet du projet.

Ensuite, dans notre fonction probas, nous avons, fait la somme sur les prédécesseur de la page se trouvant à l'indice s passé en argument, avec une disjonction de cas (celle donnée dans l'énoncé). Nous avons rajouté un autre test, if (u==0 && s==0), i.e le cas où G[s] et G[u] sont tous deux le supernoeud, car le supernoeud ne pointe pas sur lui même donc on passe à l'itération suivante.
La fonction checkProbas nous permet de vérifier ou non que la somme des probabilités vaut bien 1 (consigne de l'énoncé).

Enfin, la fonction maxTab renvoie l'indice de la page avec le PageRank le plus élevé.

Fonctions supplémentaires pour afficher les marges d'erreurs :

Une fonction absoluteValue pour pouvoir calculer des marges d'erreur.

Une fonction marginError qui remplit le tableau MarginError qui, à chaque indice i, associe la marge d'erreur lors du calcul du PageRank de la page numéro i dans le graphe.

**Etape 4 : implémentation et utilisation des matrices d'adjacence**

Pour cette étape, nous avons commencé à nous renseigner sur le principe des matrices d'adjacence sur Internet. Une fois que nous avions bien saisi le principe, nous avons commencé à rédiger la fonction creerMatAdj, qui renvoie une matrice (int **) M associée à un graphe G passé en argument. Comme le supernoeud ne pointe pas sur lui même, nous avons initialisé M[0][0]=0. Puis pour tout i,j tel que i=0 ou (c'est donc un ou exclusif) j=0, M[i][j]=1, car le supernoeud pointe sur toutes les pages et inversement. Cette fonction nous donnait de mauvais résultat, donc puisqu'elle n'était pas explicitement demandée, nous l'avons abandonné. Nous avons transformé notre graphe test wiki-nath (fichier test se trouvant dans le dossier Projet) en matrice adjacente manuellement pour tester nos fonctions.

Concernant le fichier test wiki-nath, nous l'avons crée afin de nous assurer que nos fonctions de calcul du PageRank fonctionnaient sur nos graphes. En effet, celui-ci étant petit, nous connaissions les résultats attendus.

Ensuite, nous avons réutilisé le principe de la fonction probas en la modifiant pour qu'elle agisse sur une matrice d'adjacence. De même pour refreshProbasMat et initProbaMatrix.

Concernant les tests sur les matrices données dans le dossier alea, nous avons remarqué que celles-ci ne représentaient pas des matrices d'adjacence car les coefficients diagonaux n'étaient pas forcément égaux à 1 (alors que chaque sommet pointe sur lui même) et les coefficient à la ligne ou à la colonne 0 ne respectaient pas toujours la propriété d'un supernoeud. De ce fait, nous avons apporté les modifications nécessaires pour qu'elles remplissent ces conditions.

La fonction matrixDimension ouvre le fichier et récupère la dimension de la matrice, celle ci étant essentielle pour réaliser les allocations dynamiques nécessaires.

La fonction readMatrix récupère la matrice et les probabilités attendues (celles données à la fin du fichier).

La fonction getProbaMatrix convertit les probabilités se trouvant dans le fichier de str à double.

Lors de cette étape (en testant les fichiers alea), nous avons rencontré une difficulté : la somme des probabilités valait bien 1 mais les probabilités n'étaient pas justes. Ce problème venait simplement d'une erreur de parenthèse dans la définition d'epsilon, à cause de laquelle nous avons quand même perdu une journée…

**Conclusion, ce que ce projet nous a apporté:**

Concernant la répartition des taches, Claire théorisait dans un premier temps les algorithmes, puis Nathan les mettait en pratique. Cependant, les rôles pouvaient être échangés par moments. Nous avons en effet essayé de tenir compte de la volonté de chacun, et de suivre une méthode de travail en binôme naturelle.

Pour conclure, nous voulions revenir sur ce que ce projet nous a apporté. En effet, nous avons trouvé très intéressant de devoir résoudre un tel problème par nous même car cela nous a permis d'appliquer les connaissances apprises durant le semestre, d'une façon totalement différente de celle expérimentée en TD ou en TP, puisque la réflexion à avoir était beaucoup plus poussée. En effet, nous avons parfois du faire des schéma pour visualiser les problèmes qui se posaient à nous et nous avons par moments fait face à des bugs difficiles à détecter… Tout cela nous a permis de mieux maitriser le langage C, d'apprendre ce qu'était le PageRank et surtout de nous rendre compte que toutes les connaissances que nous avons accumulé nous ont permis de produire un travail concret (ce que nous n'avons pas toujours l'occasion de faire à l'université, notamment en mathématiques).

De plus, le fait que ce travail soit à réaliser en binôme nous a beaucoup plu car nous avons du apprendre à nous faire comprendre l'un de l'autre et ainsi à mieux comprendre nous même ce que nous faisions. Par ailleurs, nous avons réussi à maintenir une bonne organisation et nous nous sommes beaucoup entraidés.