Projet de MAAIN

Moteur de recherche

Université de Paris

Ryad Azzouz,
Richard Luong,
Yassine Haouzane

March 21, 2021

Sommaire

1 Langage et environnement

Pour ce projet, on a utilisé le langage de programmation Python car il offre une certaine simplicité pour coder les différents algorithmes vu en cours malgré un manque de typage fort et d'efficacité pour des traitements des gros fichiers, peut-être aurions nous du programmer en assembleur ?

2 Corpus

Nous avons basé notre corpus sur le football français (en retenant les mots football, france, ballon) en limitant la taille de celui-ci pour limiter les temps de calculs (voir fichier corpus.py).

Pour générer notre corpus, nous avons récupéré les dumps Wikipédia en différentes parties. Puis nous avons nettoyé le fichier de base (voir fichier clean_file.py) en retirant le contenu inutile (liens externes, code css ...) et passé tout le corpus en minuscule. Enfin à partir du corpus nettoyé, nous avons généré nos différentes structures de données, CLI, relation mots-pages.

3 Forme CLI

3.1 Numérotation des pages

On commence par numéroter les pages selon l'ordre du parcours BFS. En effet, la numérotation BFS nous aide lors de la construction de la matrice sous forme CLI.

3.1.1 Génération d'un dictionnaire de positions

Voir fichier cli.py, pages_position(filename)

On commence par générer un dictionnaire nommé positions.
C'est un dictionnaire associant un titre à un couple (position de la page dans le corpus, -1). On parcourt ainsi tout le corpus avant de commencer les parcours BFS. En effet, pour le BFS nous utilisons une file contenant les titres des pages à explorer. Pour éviter de chercher l'emplacement de la page depuis le début du fichier à chaque exploration (donnant une complexité linéaire), on utilise cette index pour sauter directement. Ainsi la recherche d'une page dans un corpus est en O(1) grâce à la fonction seek(pos), qui place le pointeur directement à pos. Une critique de cette méthode pourrait être le grand nombre d'accès disque. Le deuxième élément du couple est un -1 au début. Nous laissons le suspense pour la prochaine section.

3.1.2 Premier parcours : numérotation

Voir fichier cli.py, bfs(title, r, filename)

On lance le BFS à partir d'une page principale dont son titre est passé en paramètre. Pour ce projet, nous avons choisi de le lancer sur "sport". Lorsque l'on traite une page, on va parcourir son contenu dans le corpus et ajouter les liens dans la file. Nous utilisons des expressions régulières (remerciements au passage à un excellent professeur dont le nom de famille commence par "P" et termine par "erifel" qui nous les a montré en L2 pour un cours de Langages et Automates) pour filtrer les titres dans le contenu des pages. L'index du parcours est mis en deuxième position du couple de la page considérée. C'est un parcours BFS standard sans de grandes surprises. On mentionnera tout de même que ce n'est pas un parcours à relance. Ainsi les pages non atteignables depuis la page "sport" auront pour index -1 (si elles avaient été visitées, elles auraient eu un index). En effet nous ne savions pas qu'il fallait faire le BFS avec relance, cependant vous nous avez dit que tant qu'on vous l'expliquait c'était bon. Sans relance, nous atteignons 132428 pages. Pour relancer le BFS, on pourrait chercher dans le dictionnaire de positions un titre dont l'index est -1. On pourrait arrêter de relancer lorsqu'il ne reste plus d'index égale à -1 dans le dictionnaire de positions. La fonction bfs renvoie ainsi le même dictionnaire positions mais avec l'index du parcours en deuxième élément du tuple si la page a été visitée.

3.1.3 Construction de la CLI

Voir fichier cli.py, cli(title, r, filename, dic) et add_queue_cli(c, l, i, lg, queue, dic, l_index)

La construction se fait dans un deuxième parcours du BFS. lg désigne la liste des groupes que nous avons obtenus avec l'expression régulière qui filtre les titres. Pour chaque titre trouvé, on l'ajoute à la file et on incrémente le compteur count qui représente le nombre de liens ajoutés. Après avoir trouvé tous les titres à la file, on ajoute à C (1/count) count fois, on ajoute à L la valeur de L à l'index de la dernière case où se trouve le dernier indice + count. Enfin, on ajoute à I l'index du parcours BFS de la page considérée qu'on récupère grâce à dic, le dictionnaire que nous avons généré préalablement. I est la liste des colonnes des coefficients. Avec l'identifiant Wikipédia, il aurait été plus fastidieux de déterminer la colonne d'un coefficient. Avec la numérotation BFS, c'est directement le numéro de la colonne.

3.1.4 Stockage

Pour le stockage de la CLI, voir fichier cli.py, serialize()

On écrit simplement chaque coefficient dans une ligne différente dans leurs fichiers respectifs. Leurs tailles étant raisonnables, nous n'avons pas sérialisé avec par exemple Pickle.

Pour récupérer la CLI, voir fichier cli.py, deserialize()

On ouvre chaque fichier et on charge les valeurs dans leurs listes respectives. Leurs tailles étant raisonnables, le chargement en mémoire ne pose pas de problèmes.

Pour le stockage du résultat du parcours BFS, voir fichier cli.py, bfs_serialization(positions):

On écrit dans un fichier à chaque ligne la clé et la valeur du dictionnaire positions. Chaque ligne a cette forme :
mot->(position dans le corpus, index du parcours)

Pour récupérer le résultat du parcours BFS, voir fichier cli.py, bfs_deserialization():

On ouvre le fichier contenant le résultat du parcours BFS, on reconstruit le dictionnaire du bfs à coup de splits.

3.1.5 Avantages de la CLI

Notre matrice contient énormément de zéros (elle est creuse), cela ferait beaucoup de coefficients à stocker. Par exemple si on stockait une matrice N * N avec N valant 10^6, que chaque coefficient est sur 4 octets, cela prendrait 4000 Go de mémoire. Le fait de ne pas stocker les 0 permet de réduire grandement la place car cette forme est plus compact. C'est une forme également efficace pour la manipulation de matrice pour le pagerank, d'où le L qu'on stock en plus. En effet on peut reconstruire la matrice de base sans le L. On stock d'ailleurs les valeurs sous la forme 1/degré sortant car on l'utilise pour le pagerank.

4 Dictionnaire

Pour le dictionnaire des mots nous avons pris les 10.000 mots les plus fréquents + tous les mots des titres qui ne sont pas contenus dans ces 10.000 mots. Nous avons récupéré un total de 82.310 mots.

L'utilisation de regex approprié (pas de findall par exemple qui prend énormément de temps) ainsi que le parcours de chaque ligne permet de parser tous le fichier corpus.xml aisément. Chaque ligne a été nettoyé (suppression des accents, des stop words ou mot vides ainsi que les majuscules) et on a donc récupérés tous les mots intéressants.

On récupère cette liste de mots du fichier stocké pour les traitements des différents algorithmes.

5 TF et IDF

5.1 IDF

Pour chacune des pages du corpus, nous avons calculé le coefficient IDF (Inverse Document Frequency) et stocké le calcul une fois pour toute. Comme pour le dictionnaire, on a fait un parcours du gros fichier xml et récupérer tous les mots de chaque page pour calculer l'IDF. On récupère un dictionnaire associant un mot du dictionnaire à son idf.

5.2 TF

TF ou Term Frequency est le nombre d'apparition du terme m dans le document d du corpus, le tf calculé est normalisé a l'aide du ND et stocké dans la relation mots-pages. Le tf est donc un coefficient pour nos calculs de score.

5.3 ND

ND est la norme du vecteur associé à d, nous avons comme pour l'idf et le dictionnaire fait un parcours du corpus.xml et stocker le calcul nd pour chaque page une fois pour toute dans un fichier. Cette norme nous sert en combinaison avec le TF pour l'algorithme WAND expliqué dans un autre chapitre. On récupère un dictionnaire associant le titre de la page à sa norme nd.

5.4 Relation mots-pages

La relation mots-pages est un dictionnaire associant pour chaque mot une liste de couples (pages, tf normalisé). On fait un parcours du corpus complet et pour chaque page visité on calcule le TF de chaque mot du dictionnaire et on calcule le tf normalisé c'est à dire tf(m,d)/nd(d).

```
{
    mot1: [(l1, tf1) , (l2, tf2), ..., (ln, tfn)];
    mot2: [(l2, tf1) , (l4, tf1), ..., (lm, tfm)];
    .
    .
    .
    motn: [(l3, t3) , (l5, tf5), ..., (lk, tfk)]
}
```

Le calcul de la relation mots-pages est très long, il nous a pris à peu près 11 heures sans optimisation particulière mais comme on le fait qu'une seule fois et qu'on le sérialise, le coût est plus ou moins amorti.

On a d'abord stocké une version écrite en texte brute qui fait 817 Mo, pour un dictionnaire de 82.310 mots et pour chaque mot contenant des listes de 70.000 éléments, des couples (pages, tf normalisé). La récupération en mémoire prenait un temps d'environ 10 minutes ce qui n'est pas raisonnable. On a alors eu l'idée de sérialiser le dictionnaire en format binaire pour qu'elle prenne moins de temps à charger, la bibliothèque Pickle de Python nous a été d'une grande aide.

La sérialisation et déserialisation étant fait avec cette bibliothèque, la relation mots-pages est passé de 817 Mo à 803 Mo et la déserialisation prend désormais en moyenne 1 minute à charger en mémoire au lieu des 10 minutes avec un dictionnaire écrit directement en brut.

Pour le calcul en mémoire, après une recherche, Python3 semble stocker plus de mémoire pour certain types, pour un couple (int, float) il stocke 56 octets car 40 pour un tuple + 8 octets pour chaque item soit 40 + 2 * 8 = 56. Pour une liste, Python ajoute 8 octets pour chaque élément, donc on a 56 + 8 * 30.000 = 240056 octets pour chaque liste en moyenne. Ce calcul peut être vérifié par :

```
import sys
s = [(int(40), float(3))] * 30000
print(sys.getsizeof(s)) -> donne 240056
```

La clé est un mot à peu près de 80 octets, donc on a 240056 * 80 = 19444536 octets soit 20 Mo. La représentation de la relation mots-pages en octets est de 2621536 octets, soit 2,6 Mo avec Python qui fait des optimisations, on a bien les 20 Mo.

En réalité, la relation mots-pages prend en moyenne 6 Go en mémoire. Mais Python optimise en ne chargeant pas tout le dictionnaire mais seulement sa représentation en mémoire qui est de 2 Mo (visible avec la fonction sys.getsizeof (le poids de tous les pointeurs))

6 Pagerank

6.1 Description

Le pagerank est un indice de "qualité" de page. Il indique les pages les plus populaires de notre corpus. Cette mesure nous sera nécessaire pour le traitement de requêtes. En effet, le score de fréquence tout seul n'est pas assez significatif. La mesure de pagerank permet d'indiquer ce que les humains considèrent pertinent.

6.2 Détails d'implémentation

Nous avons tout simplement adapté en Python l'algorithme que vous nous avez fourni qui permet de s'approcher de la probabilité de tomber sur la page. Nous implémentons cet algorithme pour s'approcher d'un point fixe. Calculer le pagerank sans approximation s'avère trop coûteux (prendre en compte tous les liens des pages n'est pas possible). Nous essayons donc d'approcher ce résultat en faisant des multiplications matrices, vecteur. Plus précisément en faisant à peu près 50 fois (d'après la littérature cela permet de s'approcher du point fixe) le produit de matrice, vecteur entre t(Ag) et pi (i.e. pi $\leftarrow$ t(Ag) * pi) ou t(Ag) est la transposée de la matrice CLI et pi est un vecteur de taille n initialisé a 1/n.

6.3 Réflexion passage à l'échelle

Tout d'abord le fait de garder 50 itérations dans notre algorithme suffit toujours lors d'un passage à l'échelle, car la précision de pi croit exponentiellement. Si nous avions besoin de plus de précision nous pourrions aller jusqu'à 100 itérations. La partie qui pourrait potentiellement poser problème est celle du produit matrice vecteur. Si nous devions traiter 10^{10} pages, tout d'abord stocker la CLI de 10^{10} pages ainsi que une liste de 10^{10} éléments serait compliqué. En effet, si nous avons 10^{10} pages et 5 liens par page et en partant du principe qu'un float est encodé avec 4 octets, nous devrions stocker $(10^{10} * 5 * 4)/10^9 = 200Go$. Cela demande en effet beaucoup de RAM, mais reste dans la mesure du possible. Quant au temps d'exécution, cela pourrait rester dans la mesure du possible, car le produit matrice vecteur ici est de l'ordre du linéaire de la CLI.

7 Calcul d'alpha

7.1 Description

Nous calculons alpha (et par conséquent beta car beta vaut 1 - alpha) afin de ramener à la même échelle le score de la fréquence et le pagerank. Ces coefficients sont nécessaires au bon fonctionnement de l'exécution de l'algorithme simple et de l'algorithme WAND.

7.2 Détails d'implémentation

Afin de calculer alpha, nous avons tiré 1000 requêtes aléatoires et pour chaque requête, nous avons calculé alpha. Alpha est obtenu en faisant le rapport mp/mf ou mp (resp. mf) est la moyenne des pagerank (resp. les scores de fréquence) des pages qui matchent avec la requête. Pour récupérer les pages qui matchent avec les requêtes nous avons implémenté une variante de l'algorithme simple (nous nous sommes inspiré du parcours).

7.3 Réflexion passage à l'échelle

Le passage à l'échelle pourrait poser grandement dû au fait que nous nous basons sur l'algorithme simple qui lui ne passe pas à l'échelle et aussi au fait que la taille de la relation mots-pages pourrait être très grande. Nous avions vu qu'en implémentant l'algorithme simple de traitement de requête, l'exécution de l'algorithme pouvait prendre jusqu'à 5 secondes.

8 Algorithme Simple

Pour l'algorithme simple, on fait un parcours des différentes listes des mots de la requête en cherchant les pages qui matchent entre ces listes. L'algorithme ne fait qu'un seul parcours des listes concernant les mots de la requête.

Un requête compte en moyenne 3 à 5 mots, le calcul des ces matchs prend en moyenne 1/10 secondes. Pour chacune de ces listes on effectue le calcul de score de fréquence.

Un problème rencontré a été le fait que l'on faisait des parcours linéaires pour trouver le bon index de la page dans la liste des tf pour récupérer le tf. Cela prenait énormément de temps de le faire pour toutes les pages qui ont matchées.

Une moyenne de 4000 matchs, 3 mots par requête et en cherchant pour ces 4000 matchs le tf se trouvant en moyenne à la position 30.000 sur 60.000. Avec un simple calcul pour une requête, cela nous donne 4000 * 3 * 30.000 = 360 millions de cases visités de la relation mot pages. En faisant un parcours linéaire, le calcul prenait un temps de 88s en moyenne pour avoir tous les scores.

Une solution était donc de trier la liste dans la relation mots-pages et de faire une recherche dichotomique pour résoudre le problème. On passe ainsi d'au lieu

360 millions à 90.000, ce qui est très rapide, le calcul passe donc de 88s à 0.12s en moyenne.

L'algorithme simple prend donc en moyenne moins d'une seconde pour répondre à la requête. Son avantage est qu'il est simple à programmer et pour des listes relativement petites (par rapport au vrai Internet), le temps de la requête avec cette algorithme n'est pas visible.

L'inconvénient est qu'il est très lent si les listes des pages qui matchent sont longues car on parcourt toute la liste pour calculer le score de chacune page qui a matché.

9 WAND, un cauchemard pour tant d'étudiants

9.1 Mise en place des listes à parcourir

9.1.1 Création de la structure

voir fichier wand.py, sorted_list_from_request(req, rel, pageranks)

Pour chaque mot de la requête, on va chercher dans la relation mots-pages la liste des couples (page, tf normalisé) du mot. Chaque liste sera triée par pagerank décroissant. De plus, chaque couple (page, tf normalisé) sera muni d'une troisième composante ayant une valeur égale à zéro. Cette emplacement sera dédiée au calcul du maximum.
On incorpore cette liste de tuples à une autre liste qui à la position i aura :

1. L'indice du pointeur
2. La liste des tuples (page, tf normalisé, maximum (0 au début)) du mot
3. La taille de la liste des couples
4. Le mot

La structure finale est une liste englobant toutes ces listes.

9.1.2 Génération du maximum

voir fichier wand.py, maximum_all(all_lists)

Pour chaque mot m, on veut parcourir de droite à gauche la liste des tuples et calculer le maximum entre IDF(m) x TF(m, d) normalisé vues jusqu'à présent. Cette valeur du maximum est stockée en troisième position du tuple.

9.2 Génération du top K initial

voir fichier wand.py, fill_top_k(top_k, all_lists, pageranks, max_k)

Le top K est un tas dans lequel on va mettre les max_k meilleurs pages. La stratégie pour initialement le remplir est la suivante. On regarde les valeurs pointées dans chaque liste de la structure. Parmi elles, on ajoute la page ayant le plus grand pagerank dans le top K sous la forme d'un couple (score total de la page, page). On avance ensuite le pointeur associé à la liste de la page ajoutée. On répète cette stratégie jusqu'à avoir max_k pages dans le tas. Si on se rend compte qu'une liste est vide pendant l'opération, elle sera retirée. On évite également d'ajouter des doublons dans le top K, si la page est déjà présente et que le nouveau score est plus grand, on le met à jour. Lorsque le top K est rempli ou que les listes sont vides, le gamma est le plus petit score total du tas.

9.3 Déroulement de l'algorithme

voir fichier wand.py, wand(request, pageranks, relation, alpha))

On commence par trier les listes par pointeurs décroissants. On calcule le pagerank de la page pointée dans la première ligne. Ce pagerank va nous aider à trouver l'indice de la liste où se trouve le pivot. Pour trouver le pivot, on applique votre formule décrite dans le TP3, malheureusement mes compétences en Latex ne sont pas assez poussées pour la reproduire ici. On vous convie cependant à regarder l'implémentation dans la fonction find_pivot_last_index. Si on ne trouve pas de pivot, on renvoie le top K car les pages restantes ne pourront jamais espérer rentrer dans le top K. Sinon, on incrémente les pointeurs des listes inférieurs au pivot jusqu'à trouver une page ayant un pagerank inférieur ou égal à celui du pivot. Pour faire cela, on procède par recherche dichotomique. Si l'on considère que l'on a suffisamment de mots de la requête, on calcul le score total du pivot. Si il dépasse gamma, on met à jour le top K en l'ajoutant et en retirant la page au score minimale du tas. On met à jour gamma qui devient la nouvelle valeur du score total le plus petit du top K. On avance les pointeurs des pages qui pointent vers le pivot. Sinon on renvoie le top K car les pages restantes ne pourront jamais espérer rentrer dans le cercle très fermé du top K.

9.4 Avantages par rapport à l'algorithme simple

L'algorithme est plus rapide et s'arrête beaucoup plus vite. En effet, on ne garde que les k meilleurs pages. On se sert également du gamma pour sauter plus loin, dans l'algorithme simple on saute moins loin. De plus, on sait qu'au bout d'un moment, aucune page ne pourra rentrer dans le top K, on pourra ainsi finir l'algorithme très vite. Avec l'algorithme simple on parcourt jusqu'à la fin. On fait également moins de calculs du score total car on sait que ce n'est pas la peine. On peut également sélectionner une partie des mots de la requête et c'est moins pénible à faire avec WAND.

9.5 Inconvénients de l'algo WAND

Le précalcul du vmax prend du temps et il faut le stocker. Seuls les k premiers résultats sont exacts, on a pas de garanties sur les autres. L'implémentation de cette algorithme favorise la dépression du programmeur. Votre aide a été précieuse pour tenir le cap, vous êtes quelqu'un de bien.

10 Serveur

10.1 Description

Pour le serveur, nous avons mis en place un serveur HTTP simple sur le port 8000. Il consiste en un simple formulaire qui attend la requête. Lorsque l'on soumet une requête, nous traitons la requête ce qui nous renvoie les ID des pages (du parcours BFS).

10.2 Détails d'implémentation

Sachant que nous récupérons les ID du BFS, nous devons récupérer les titres de ces pages et faire la traduction ID BFS ID Corpus afin de pouvoir générer les hyperliens vers les pages Wikipédia. Pour récupérer les titres à partir des ID BFS, nous avons un dictionnaire en mémoire ID BFS $\rightarrow$ (titre, ligne dans le fichier corpus). Pour récupérer les ID initiaux, nous avons utilisé les lignes dans le fichier des corpus et nous sautons à la ligne (à l'aide d'un appel de seek(n) qui est en O(1)) correspondante dans le fichier du corpus et nous récupérons l'id de la page qui correspond. Lorsque nous avons l'ID du corpus et le nom de la page avec cela, nous pouvons afficher le nom de la page avec son URL qui est de la forme : https://fr.wikipedia.org/?curid=[ID DU COPUS].

10.3 Réflexion passage à l'échelle

Cette manière de générer les liens fonctionne plutôt bien (en dessous d'une seconde pour WAND) même s'il faut garder à l'esprit que les accès sur disque sont beaucoup plus long que les accès en mémoire.

Nous aurions pu essayer de garder toutes les informations nécessaires en mémoire, ce qui aurait demandé plus de RAM (cependant il faut mentionner que nous étions déjà aux limites de nos machines avec la relation mots-pages qui pesait entre 6 Go et 10 Go en fonction de la machine).

Cette façon de faire aurait par contre du mal à passer à l'échelle, car un nombre de pages plus élevé impliquerait soit une utilisation de la RAM plus intense ou un nombre plus élevé d'accès au disque.

Pour remédier à cela nous aurions pu nous baser sur les id Wikipédia même si cela aurait complexifié le projet.