Devoir N° 2 – Implémentation d'un moteur de recherche

Année 2019-2020

Un moteur de recherche, tel que par exemple Google ou Qwant, permet de trouver rapidement et de manière optimisée des ressources à partir d'une requête. La procédure qui permet à un moteur de recherche d'optimiser le temps nécessaire à l'exécution d'une requête est l'indexation automatique de documents, qui consiste à réorganiser des documents afin d'accélérer ultérieurement la recherche en se basant sur un index inversé. Un index (dictionnaire) inversé est une correspondance entre le contenu d'un ensemble de documents (les mots qui les constituent), et sa position dans l'ensemble des données. Autrement dit il associe chaque mot (ou terme) aux documents dans lesquels il apparaît. La Figure 1 illustre un exemple d'index inversé associé à un ensemble de documents.

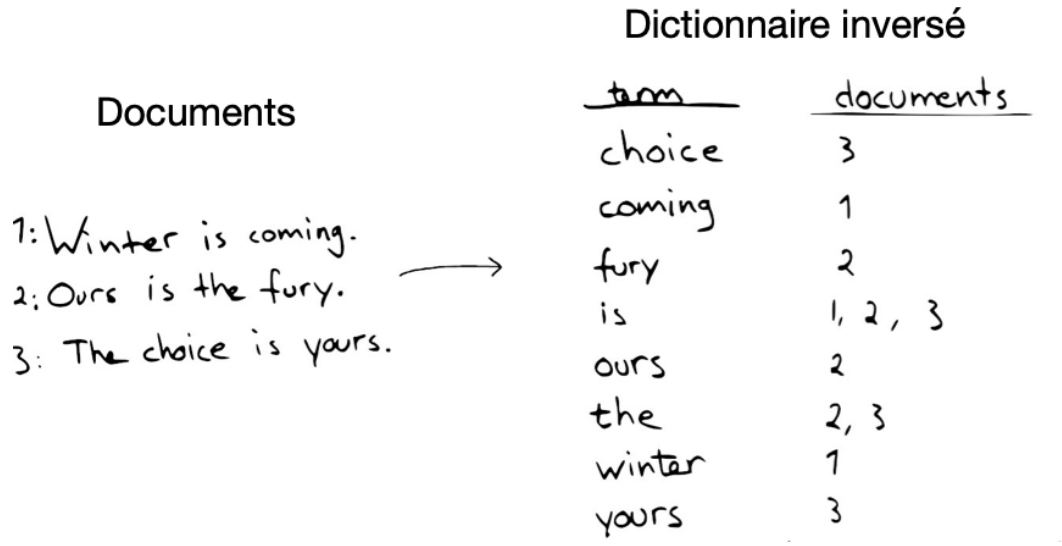

Figure 1. Exemple d'index inversé associé à un ensemble de 3 documents

Ce TP permet de créer un moteur de recherche basique implémentant une procédure d'indexation de documents.

Partie 1 – Documents représentés par des listes

On considère un moteur de recherche implémenté avec la classe `Moteur_lists` dont la spécification partielle est donnée dans le fichier `devoir2.py` déposé dans votre espace de travail sur Moodle. La classe `Moteur_lists` a comme attributs deux dictionnaires :

- **inv_indx** : dictionnaire qui représente l'index inversé où les clés représentent les mots et les valeurs représentent les documents sous la forme d'un ensemble `set()`. Par exemple **inv_indx**`["is"]` est l'ensemble `{1,2,3}`.
- **docs_links** : dictionnaire qui a pour clés l'identifiant d'un document (un entier) et pour valeur la variable représentant ce document. Dans cette première partie on considèrera que le document est représenté par une liste de mots. Par exemple si on considère le document 1 de nom `doc1` et d'identifiant 1 contenant les termes 'Winter', 'is', 'coming', on aura : `doc1 = ['Winter', 'is', 'coming']` et **docs_links**`[1] = doc1`

Dans l'implémentation, vous utiliserez les structures de données prédéfinies en Python : dictionnaires (`dict`), ensembles (`set`) (voir la documentation sur les ensembles : `https://docs.python.org/3.8/tutorial/datastructures.html#sets`).
Vous pourrez utiliser les méthodes `union` et `intersection` associées aux ensembles.

1. Écrivez la méthode `add_to_index(self, docname)` qui permet d'indexer un document donné de nom `docname` qui contient une liste de mots. Vous considèrerez que le nouveau document est placé à la fin du dictionnaire `docs_links` (en incrémentant le numéro identifiant le document). De plus, vous mettrez à jour le dictionnaire `inv_indx` pour chacun des mots entrés :
   - Si le mot existe déjà, vous adjoindrez à l'ensemble des documents qui lui est associé l'identifiant du nouveau document `docname`.
   - Si le mot n'existe pas vous créerez une nouvelle entrée dans la table `inv_indx`.

   Par exemple, si on considère un 4ème document : `doc4 = ['Winter','is','here']`, et que l'on appelle la méthode `add_to_index` appliquée à ce document pour le moteur `my_engine` : `my_engine.add_to_index(doc4)`
   on modifiera les dictionnaires ainsi :

   - Dans le dictionnaire `docs_links`, on créera une nouvelle clé (identifiant 3) avec la valeur associée `doc4`.
   - Dans le dictionnaire `inv_indx`
     - à la clé 'Winter' sera associé l'ensemble `{1,4}`
     - à la clé 'is' sera associé l'ensemble `{1,2,3,4}`
     - à la clé 'here' sera associé l'ensemble `{4}`
2. Écrivez la méthode `search_word(self, w)` qui permet de renvoyer l'ensemble (de type `set`) des identifiants des documents dans lesquels le mot `w` apparaît.
   Par exemple, pour notre moteur `my_engine`, on a :
   `my_engine.search_word('the') = {2, 3}`
3. Écrivez la méthode `search_inverse_word(self, w)` qui permet de renvoyer l'ensemble des identifiants des documents qui ne contiennent pas le mot `w`. Par exemple :
   `my_engine.search_inverse_word('the') = {0}`
4. Écrivez les méthodes :
   (a) `and_word(self, w1,w2)` qui permet de renvoyer l'ensemble des identifiants des documents qui contiennent `w1 et w2`. Le `et` correspond ici à l'intersection des ensembles d'identifiants de documents.

(b) `or_word(self,w1,w2)` qui permet de renvoyer les documents qui contiennent `w1 ou w2`. Le `ou` correspond ici à l'union des ensembles d'identifiants de documents.

5. On veut rechercher les documents qui contiennent (ou pas) des mots, en respectant des formules logiques. La recherche s'exprime donc comme une expression logique écrite avec des opérateurs logiques. On considèrera ici les opérateurs ET (intersection), OR (union), NOT (non appartenance).

   - **Exemple 1**. Soit la recherche de documents respectant l'expression logique : `("droit AND pénal") OR ("droit AND civil")`. Dans notre moteur de recherche les expressions seront écrites en notation polonaise inverse ce qui donne pour l'expression ci-dessus :
   `"droit pénal AND droit civil AND OR"`
   - **Exemple 2**. La recherche de documents contenant "droit" mais pas "numérique" peut s'exprimer en logique : `"droit AND NOT numérique"`, ce qui donne en notation polonaise inverse : `"droit NOT numérique AND"`

   Écrivez la méthode `eval_exp_terms(self, exp)` qui évalue l'expression logique `exp` et retourne l'ensemble des documents respectant cette formule logique. Les expressions seront des chaînes de caractères composées de mots et d'opérateurs (AND, OR, NOT).
   Indication Vous pourrez vous appuyer sur la modélisation des piles (classe `Pile_List`) vue dans le TP 7, afin de simuler l'évaluation d'expressions logiques.

6. Testez l'évaluation d'expressions booléennes exploitant des mots et des opérateurs (AND, OR, NOT), en créant vos propres documents (listes de mots), et en proposant une variété d'exemples.

Partie 2 – Documents représentés par des fichiers – Optionnel

Dans cette partie on s'intéresse à la manipulation de fichiers en python. Pour travailler avec le contenu des fichiers (lire ou écrire), il faut "ouvrir" ces fichiers avec la fonction `open()`. Cette fonction prend obligatoirement deux arguments : le chemin du fichier (relatif ou absolu), et le mode d'ouverture qui défini si on va lire le contenu du fichier ou écrire dans le fichier.

Exemple : `fichier = open('/chemin vers le fichierfichier.txt', 'r')` ». Pour être en mode « lecture », on utilise « r » pour du texte, « rb » pour une lecture binaire ; pour être en mode écriture on utilise « w » pour écrire du texte, « wb » pour une écriture binaire. Lorsque le fichier est ouvert, pour lire son contenu on utilise la méthode `read()` qui retourne le contenu du fichier. Exemple : `contenu = fichier.read()`.
Pour écrire dans le fichier on utilisera la méthode `write()`. Exemple : `fichier.write(content)`. Le fichier ouvert étant verrouillé, il ne faut pas oublier de le fermer avec la méthode `close()` après la lecture ou l'écriture.

Au niveau de l'implémentation, vous réaliserez le programme dans un fichier différent de celui de la partie 1. Vous serez amenés à compléter la spécification donnée dans le fichier `devoir2.py`.

Le moteur de recherche est représenté par la classe `Moteur_files`, dans laquelle chaque document indexé est identifié par un entier et caractérisé par un nom de fichier. Ainsi, dans le dictionnaire **docs_links**, à l'identifiant du document `fid` on fera correspondre une référence à un fichier qui sera représentée par le nom de fichier `fname` : `docs_links[fid] = fname`.

Remarque. On pourra mettre la valeur de `docs_links[fid]` à None si le document n'est pas un fichier.

1. Testez la méthode `tokenize(self,s)` qui à partir d'une chaîne de caractères renvoie une liste de termes.

2. Écrivez les méthodes :

   (a) `add_to_index_file(self,fname)` qui permet d'indexer un document donné sous la forme d'un fichier texte.
   Indications 1. Vous procéderez comme pour la fonction `add_to_index(self, docname)` de la partie 1 sauf que le document représenté par la variable `docname` est remplacé par le nom de fichier `fname`.
   2. Vous utiliserez la fonction `tokenize(self,s)` qui permet de transformer le contenu du fichier en une liste de termes.

   (b) `add_bulk_files(self, filenames)` qui permet d'indexer une liste de documents représentés par une liste de noms de fichiers `filenames`.
   Remarque Lorsqu'il y a beaucoup de fichiers, il est possible de lire la liste des fichiers contenus dans un répertoire `rep` en utilisant la bibliothèque `glob` qu'il faudra alors importer et l'instruction :

   ```
   filesList = glob.glob("rep/*.txt")
   ```

3. Écrivez les méthodes suivantes (compléter la spécification) :

   (a) `intersection_sets_terms(self,terms)` qui renvoie les documents qui résultent de l'intersection d'une liste de termes.

   (b) `union_sets_terms(self,terms)` qui renvoie les documents qui résultent de l'union d'une liste de termes.

   (c) `intersection_inv_sets_terms(self,terms)` qui renvoie les documents qui ne contiennent pas une liste de termes.

   (d) `intersection_sets(self, sets)` qui renvoie les documents qui résultent de l'intersection d'une liste de `set`.

4. Écrivez la méthode `search(self, must_include, at_least_one, exclude)` qui retourne l'intersection des documents qui contiennent tous les termes dans la liste `must_include` avec ceux qui contiennent au moins un des termes dans la liste `at_least_one` et ceux qui ne contiennent aucun des mots de la liste `exclude`.
   Indication Vous utiliserez la méthode `intersection_sets(self, sets)` définie à la question précédente.

5. Écrivez la méthode `print_result(self, search_result, max_results)` qui permet d'afficher, pour chacun des documents dont les identifiants sont contenus dans `search_result`, l'identifiant du document, son nom (chemin pour accéder au fichier) et son contenu (affichage du texte). Le nombre de documents à afficher est déterminé par l'argument `max_results`.