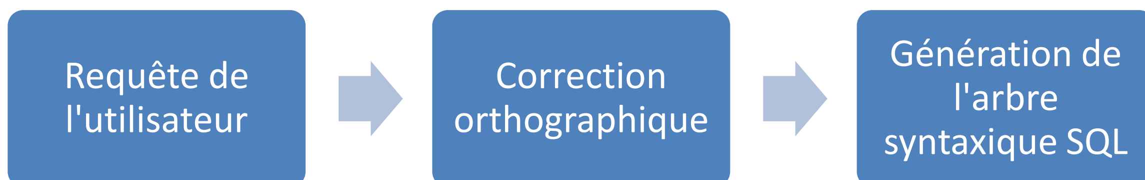
Projet LO17
Analyses morphologique et syntaxique

Introduction

Le projet de LO17 consiste en un archivage des pages d'information du site LCI (issues de la rubrique « Monde » et récoltées entre le 25/02/2005 et le 02/03/2006). On souhaite ensuite pouvoir indexer ces données afin de pouvoir par la suite effectuer des requêtes en langage naturel, à la manière d'un moteur de recherche tel que Google, pour obtenir des informations sur les articles à partir de critères tels qu'une date, un thème, un contenu ou un lieu par exemple.

Dans un premier temps, il nous a fallu préparer le corpus documentaire qui a été récupéré sous forme de pages HTML et l'indexer via des fichiers inverses permettant de décrire les documents à l'aide de tout ou partie des éléments contenus dans le corpus : ceci a été détaillé dans le premier rapport du projet.

Dans un second temps, nous avons traité l'aspect de l'analyse morphologique par le biais du développement d'un correcteur orthographique, permettant de tolérer quelques erreurs de frappe ou d'orthographe entrées malencontreusement par l'utilisateur lors de la saisie des termes de la recherche. L'analyse syntaxique a quant à elle permis de générer une grammaire qui permet à l'utilisateur de saisir une requête en langage naturel (langue française) qui sera ensuite convertie en langage de recherche SQL.



Analyse morphologique

L'analyse morphologique consiste à développer un mécanisme permettant de généraliser des termes en un lemme unique afin d'élargir le champ de recherche et de ne pas se retreindre à une terminaison unique d'un mot ni à une orthographe unique d'un mot à rechercher.

Exemple : si l'utilisateur entre le mot « chantent » et que les documents contiennent le lemme « chante », ils ne seront pas retournés (on augmente considérablement le silence en terme d'efficacité).

Autre exemple : si l'utilisateur entre le mot « chnate » (inversion de 2 lettres au clavier), les documents contenant le lemme « chante » ne seront pas non plus retournés (silence).

Afin d'éviter ces deux problèmes, il est apparu nécessaire de développer un module de correction orthographique regroupant les deux notions vues précédemment, soient la « racinisation » des mots (plusieurs dérivations d'un même mot regroupées au sein d'une même racine) et la correction orthographique (fautes de frappe, fautes d'orthographe, etc...).

Lorsque l'utilisateur entre une requête en langage naturel afin de rechercher un document, le module d'analyse morphologique est exécuté, afin de retourner une suite de lemmes normalisés (racines des mots, corrections des erreurs d'orthographe, etc...) qui sera ensuite passée à l'analyseur syntaxique afin de générer la requête SQL et de récupérer les documents correspondant à la requête initiale.

Dans un premier temps, le module de correction orthographique met tous les caractères de la requête en minuscules, afin d'éviter les problèmes liés à la casse (pas de différence entre majuscules et minuscules).

Ensuite la requête est traitée comme un flux de mots séparés par des espaces (problèmes posés par les apostrophes, les traits d'union, etc... qui sont donc gérés comme un seul mot ex : « d'article » est considéré comme un mot unique, « chausse-pied » également...).

Chaque mot est ensuite comparé à ceux du dictionnaire (liste des mots récupérés dans les documents avec leur lemme associé) : s'il y est présent, c'est qu'il n'y a pas d'erreur d'orthographe et que le mot est connu tel quel au sein du dictionnaire, donc on récupère son lemme associé pour normaliser la requête. Dans le cas contraire (cas fréquent car le dictionnaire ne peut pas connaître toutes les dérivations de tous les mots du corpus ni toutes les fautes de frappe possibles !), l'algorithme de recherche par préfixe va s'exécuter sur le mot en question.

Algorithme de recherche par préfixe

Le principe de cet algorithme est de comparer le nombre de lettres communes entre le début du mot et le début d'un mot du lexique (pour tous les mots du lexique) et si ce nombre est assez grand, cela signifie que le mot est probablement une dérivation du mot du lexique (racine) : on stocke ainsi le lemme associé dans une liste de lemmes candidats. La valeur associée à chaque lemme candidat est une valeur de proximité calculée comme suit : nombre de lettres communes entre le mot et le mot du lexique / longueur moyenne des deux mots (on a choisi de prendre la moyenne car prendre le

maximum comme dans le cours pénalise trop les racines assez longues comme « eraient », « erions », etc...).

Une fois tous les mots du dictionnaire testés, on retourne le meilleur des lemmes candidats grâce à un mécanisme double :

- On ne conserve que les lemmes candidats qui ont la valeur maximale à un écart δ près (δ fixé arbitrairement à 0.15 après test)
- On ne conserve que les lemmes qui ont un nombre de lettres communes maximal avec le mot de la requête

On retourne ensuite le lemme qui a la valeur maximale au sein de la sous-liste des meilleurs lemmes ou bien le premier de la liste s’il y a conflit.

Nous avons opté pour renvoyer un seul lemme car c’était plus simple pour notre application mais on aurait pu imaginer un système qui renvoie la liste des meilleurs lemmes candidats et les propose à l’utilisateur afin qu’il sélectionne la requête appropriée à sa recherche (système de suggestions à la Google).

Les différents seuils fixés pour l’algorithme de recherche par préfixe sont les suivants (ils ont été fixés après test de façon arbitraire) :

- Longueur minimale des mots à comparer fixé à 3 lettres, c'est-à-dire que seuls les mots contenant plus de 3 lettres sont comparés par l’algorithme (les mots de moins de 3 lettres n’ont pas vraiment de racine...)
- Ecart maximal entre les longueurs des deux mots à comparer fixé à 5 lettres, c'est-à-dire qu’il faut que les deux mots à comparer aient une différence de tailles de 5 caractères au maximum.
- Nombre minimum de lettres communes entre le début des deux mots comparés fixé à 4 caractères : les deux mots doivent donc avoir au moins les quatre premières lettres communes (idée de même racine de deux mots)
- Seuil à partir duquel les lemmes sont considérés comme lemmes candidats fixé à 0.6 : la proximité (calcul expliqué plus haut) entre le mot de l’utilisateur et le mot du lexique doit être au moins de 0.6 pour que le lemme correspondant au mot du lexique soit ajouté à la liste des lemmes candidats. On a fait le choix de prendre un taux assez élevé afin de favoriser l’algorithme de Levenshtein qui est plus généraliste que l’algorithme de recherche par préfixe.

Exemple de requête avec le mot « chrétienté » :

saisie : chrétienté
algo prefixe={chrétien=0.8} → signifie que la proximité entre « chrétienté » et « chrétien » est de 0.8
Requête corrigée :
chrétien

Si aucun lemme n'est considéré comme résultat satisfaisant par rapport aux seuils définis, le mot va être soumis à l'algorithme de Levenshtein qui va s'occuper de trouver les lemmes du lexique qui sont les plus proches (en terme de distance) du mot entré par l'utilisateur.

Algorithme de Levenshtein

Cet algorithme consiste à calculer la distance entre deux mots différents. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. Une fois cette distance récupérée, on calcule un coefficient de corrélation entre les deux chaînes grâce à la formule suivante : 1 – (Distance de Levenshtein / longueur maximale des deux chaînes).

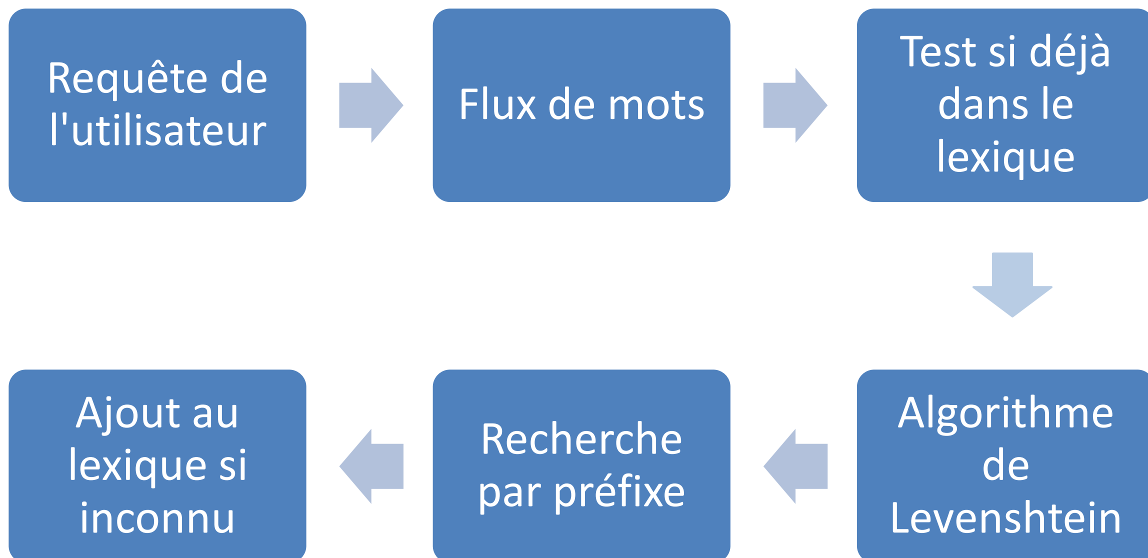
Si ce coefficient est supérieur à un seuil que l'on a défini par test à 0.6, on ajoute le lemme correspondant au mot du lexique comparé dans la liste des lemmes candidats.

Ensuite, le même mécanisme que pour l'algorithme de recherche par préfixe est appliqué afin de récupérer le meilleur lemme candidat possible (on favorise les lemmes qui ont un nombre de lettres communes maximal avec le mot entré par l'utilisateur => recherche de permutation ou de préfixe à l'intérieur du Levenshtein).

Exemple de requête avec le mot « chrétienté » contenant une erreur de frappe :

saisie : chértienté → erreur de frappe (inversion de 2 lettres au clavier = erreur commune)
Meilleur lemme candidat (Levenshtein) : {cherch=0.6, charité=0.6, chrétien=0.6, croient=0.6}
→ les lemmes candidats ont tous la même proximité avec le mot entré par l'utilisateur (0.6) donc on prend celui qui contient le plus grand nombre de lettres communes avec le mot entré par l'utilisateur, soit le lemme « chrétien »
algo levenshtein={chrétien=0.6}
Requête corrigée :
chrétien

Enfin, si aucun lemme ne correspond aux exigences en terme de proximité par rapport aux seuils fixés, le mot de l'utilisateur est laissé tel quel dans la requête (on peut imaginer qu'on enrichirait le lexique avec ce mot pour une éventuelle future requête contenant ce même mot). Cela arrive notamment lorsque le mot contient trop de fautes (d'orthographe ou de frappe) ou que le lexique est trop pauvre (d'où la nécessité de l'enrichir dynamiquement).



On aurait pu éventuellement introduire d'autres mécanismes tels que la détection de fautes de frappe par rapport à la position des touches sur le clavier mais cela n'a pas été implémenté faute de temps.

Exemple : si on tape la lettre 's' sur un clavier AZERTY, il y a de fortes chances pour que la lettre qu'on ait voulu taper appartiennent à la liste {'a', 'z', 'e', 'q', 's', 'd', 'w', 'x'} car ce sont les touches du clavier qui sont à côté de la touche 's' (il faudrait donc tester toutes les permutations possibles avec ces lettres).

Analyse syntaxique

L'objectif de cette seconde partie est de générer l'équivalent en SQL de la requête, une fois qu'elle a été normalisée (lemmatisée) et corrigée grâce à l'étape de l'analyse morphologique.

L'étape de l'analyse syntaxique se divise en deux parties :

- Lexicalisation de la requête : le lexer contrôle l'orthographe des mots de la requête à partir de la liste des tokens que nous lui avons spécifiés.
- Analyse de la grammaire des tokens générés par le lexer : le parser s'assure que ces tokens sont arrangés en des constructions grammaticales valides (nous expliquerons par la suite les structures grammaticales que nous avons spécifiées au parser), et puis générera à partir de ces tokens un arbre syntaxique (en SQL) pour ordonner la requête et lui donner un sens.

Dans un premier temps, nous avons essayé de produire une grammaire qui sera en mesure de reconnaître les questions (requêtes) les plus récurrentes.

Afin de déterminer celle-ci, nous avons identifié les types de questions les plus récurrents à partir d'un corpus de 241 questions en langage naturel concernant les informations présentes dans les pages LCI.

Après analyse du corpus de questions, nous avons choisi de répondre aux quatre types de questions suivantes:

Introduction de question	Partie recherchée	Pivot	Sujet
Je veux	tous les articles	qui parlent	de Bush et usa
Retourner	la liste des articles	écrits par	Faubert
J'aimerais consulter	les articles	liés à	la Belgique
Je voudrais	la liste des pages	qui parlent de	George
Donnez-moi	les résumés de tous les textes	traitant	la politique

Introduction de question	Partie recherchée	Pivot	Sujet	Date
Je veux	la liste des articles	parus en		avril 2005

J'aimerais	les articles	parus		le 15 février 2005

- Questions portant sur un thème précis :

Introduction de question	Partie recherchée	Pivot	Thème	Sujet
Quels sont	les articles		sur le thème	de l'économie

- Questions nécessitants un calcul :

Introduction de question	Partie recherchée	Pivot	Sujet	Date
Combien	d'articles	traitent	de l'Irak	
Combien	de pages	traitent	du procès de Mickael Jackson	en 2005

Le choix de ces types de questions impose l'élimination des questions portant sur le tri telles que : « Je veux les 10 derniers articles qui traitent de la crise économique », « Quels sont les auteurs qui ont écrit le plus d'article ce mois-ci ? »…,

Dans un second temps, il a fallu supprimer des mots de la requête qui sont susceptibles de perturber la structure de la question telle que l'on a définit précédemment, afin de répondre aux différentes variantes de la question posée.
Pour cela, on a choisi de supprimer tous les mots de liaison de la langue française qui n'apportent pas de sens à la requête.
Exemples des mots supprimés : « je, j', le, la, les, l', d', qui, par, en, dont, tous, toutes, … ».

Ensuite, nous avons déterminé les groupes de mots d'introduction, de parties recherchées et de pivots afin de les associer à un symbole Lex.
Après identification des groupes de mots associés à la partie recherchée (articles, liste des articles, themes, listes des resumes…), on a trouvé que la recherche peut porter sur cinq Lex : les articles, les titres, les thèmes, les pages et les auteurs. Ensuite on a associé à chacun de ces Lex le groupe de mots qui lui est associé (par exemple les valeurs « article, liste des articles, ensemble des articles, … » sont associés au Lex : article).

Afin d'identifier les différents Lex associés aux pivots, il a fallu analyser le critère de la recherche de la question, par exemple :

- Je voudrais la liste des articles écrits par dstrauss@tf.

Le pivot « **écrits par** », nous indique qu'on cherche une adresse mail (un auteur).

- Je voudrais tous les articles parus le 22 février 2005.

Le pivot « **parus le** », nous indique qu'on cherche une date.

- Je voudrais la liste des articles sur le thème du pape.

Le pivot « **sur le thème** », nous indique qu'on cherche un thème précis.

- Quels sont les articles traitant du Moyen-Orient ?

Le pivot « **traitant** + mot », nous indique qu'on cherche les articles qui contiennent « mot » dans leur contenu, de même pour les pivots : contient, contiennent, parlent, parlant, traitent, …

La syntaxe de la requête nous donne des indications quant à la ou les tables de la base de données où l'on va chercher les informations (clause SQL « FROM »), ainsi que les conditions de filtrage (clause SQL « WHERE »).

Select	Partie recherchée	From titreresume	Where
J'aimerais consulter	la liste des articles	liés à	la Belgique
Je voudrais	les articles	qui parlent	du pape

La phase d'analyse syntaxique a nécessité d'enrichir le dictionnaire (le lexique) utilisé par la phase d'analyse morphologique afin de corriger les fautes de frappe et d'orthographe entrées par l'utilisateur, car il fallait que les lex utilisés dans l'analyse lexicale soient reconnus par l'analyse morphologique et donc soient présents dans le dictionnaire afin de ne pas être modifiés lors de la correction orthographique, ce qui pourrait sinon entraîner une non-détection des bons lex et donc une erreur lors de la génération de l'arbre SQL.

Construction de l'arbre :

On a défini dans notre analyse grammaticale, un arbre qui représente la requête. Il est construit au fur et à mesure de l'analyse de la requête et est représenté par la classe Java « Arbre », fournie lors des TD.

La phase d'analyse lexicale s'occupe de supprimer les mots de liaison et de faire correspondre aux différents mots de la requête les tokens entrés dans le lexer :

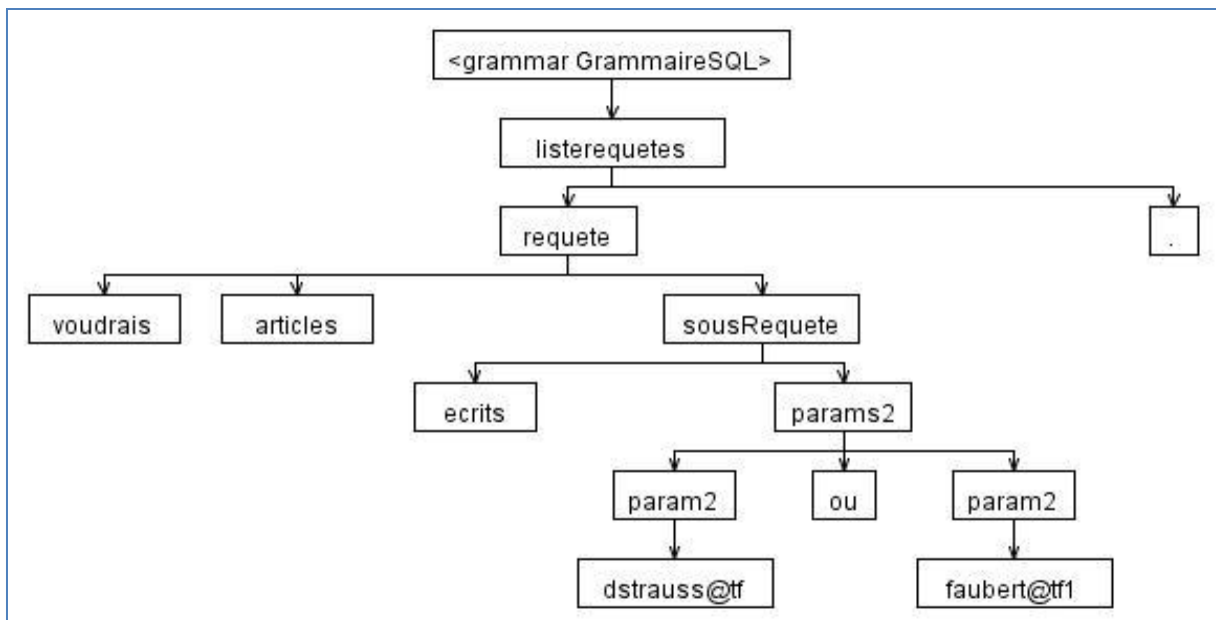
« je voudrais tous les articles ecrits par dstrauss@tf ou par faubert@tf1. »

Lexer

« voudrais articles ecrits dstrauss@tf ou par faubert@tf1. »

Ensuite, l'analyse grammaticale s'occupe de générer l'arbre syntaxique (AST) en fonction de la structure grammaticale de la requête et des différentes règles entrées dans l'analyseur grammatical (via l'outil AntLR).

La figure ci-dessous représente la structure d'arbre correspondant à la requête : « je voudrais tous les articles ecrits par dstrauss@tf ou par faubert@tf1. » :



Le premier niveau de l'arbre contient 3 nœuds :

- « voudrais » correspond à la clause « select ».
- « articles » correspond à la source recherchée « article ».
- « sous requête » correspond aux critères de la recherche, sa profondeur dépend du nombre de conjonctions (et/ou) présentes dans la requête.

Lors de la création de l'arbre syntaxique, les différentes règles grammaticales créent l'arbre (objet de la classe « Arbre ») SQL correspondant à la requête.

Exemple de construction de l'arbre SQL pour la requête complète entrée par l'utilisateur :

```
requete returns [Arbre req_arbre = new Arbre("")]
        @init {Arbre ps_arbre; int test = 0;} :
                (SELECT
                        {
                                req_arbre.ajouteFils(new Arbre("","select distinct"));
                        }

                (ARTICLE
                        {
                        req_arbre.ajouteFils(new Arbre("","article"));
                        test = 0;
                        art_page = "article";
                        }
                 | PAGE
                        {
                        req_arbre.ajouteFils(new Arbre("","page"));
                        test = 1;
                        art_page = "page";
                        }
                 | AUTEUR
                        {
                        req_arbre.ajouteFils(new Arbre("","email"));
                        test = 2;
                        art_page = "";
                        }
                 | TITRE
                        {
                        req_arbre.ajouteFils(new Arbre("","mot")); // mot dans table titre
                        test = 3;
                        art_page = "";
                        }
                )
```

Cette règle renvoie donc un objet Arbre ; lors de la détection du lex « SELECT », on rajoute « select distinct » à l'arbre SQL que l'on est en train de générer. Puis en fonction de ce que l'on détecte (en fonction de ce que l'utilisateur désirait rechercher) on ajoute à l'arbre SQL « article », « page », « email » ou « mot » (pour le titre car la table PostgreSQL « titre » a pour clé « mot »).

Lors du parcours de la requête de l'utilisateur, l'arbre SQL se construit et à la fin de l'analyse, il est remonté sous forme de chaîne de caractères dans la règle « listerequetes » (pivot des règles) et donc est affiché ensuite dans le programme Java qui appelle les analyses lexicale et syntaxique :

Par exemple, la chaîne SQL correspondante à la requête: « je voudrais tous les articles ecrits par dstrauss@tf ou par faubert@tf1. » est *(**select distinct** article **from** email **where** (((email = 'dstrauss@tf') **OR** (email = 'faubert@tf1')))).*

Cette chaîne représente donc la requête SQL équivalente à la requête entrée en langage naturel par l'utilisateur. Cette requête SQL subira ensuite un post-traitement afin de respecter scrupuleusement la syntaxe SQL et supprimer les éventuelles indéterminations avant d'être exécutée sur les tables PostgreSQL afin de récupérer les résultats de la requête.

Conclusion

Ces deux phases d'analyses morphologique et syntaxique ont permis de mettre en pratique :

- L’utilisation d’un lexique de mots avec leurs lemmes correspondants.
- Différentes techniques de correction d’erreurs (recherche par préfixe, Levenshtein, position des touches sur un clavier, etc...) afin de tolérer des faute d’orthographe et de frappe de la part de l’utilisateur quand il saisit une requête.

  Cependant, les algorithmes de préfixe et Leveishtein, ne sont pas assez fiables, ils peuvent proposer à l’utilisateur des corrections qui ne sont pas conformes de ce qu’il cherche.
- Transcrire une phrase d’un langage à un autre en utilisant les grammaires et un outil type AntLR et plus précisément la génération d’une requête en format SQL à partir d’une requête entrée par un utilisateur en langage naturel.
- L’enchaînement des phases d’analyse : analyse morphologique puis analyse lexicale puis analyse syntaxique.

La phase d’analyse syntaxique (la grammaire) est toujours en cours d’évolution ; en effet, la grammaire va s’enrichir au fur et à mesure jusqu’à la fin du projet afin de permettre à l’utilisateur d’entrer un nombre plus conséquent de requêtes différentes, avec des structures de plus en plus évoluées, même s’il paraît évident que toutes les formes de requêtes ne pourront pas être implémentées mais on essaiera de rendre le système de requêtage le plus complet et le plus robuste possible.