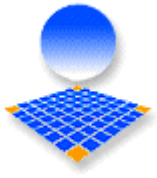
Ministère de l'enseignement supérieur
et de la recherche

Université de Montpellier II

Rapport de Projet Informatique ULIN 607
de la Licence informatique 3ème année effectué
du 25 février 2010 au 10 mai 2010

# GÉNÉRATEUR D'URL ÉQUIVOQUES

DESBUISSONS Nicolas
DIAKONOV Mikhail
FONQUERNIE Éric
HEUZÉ Stéphane



PROJET ENCADRÉ PAR PHILIPPE GAMBETTE

# Remerciements

Nous tenons à remercier notre tuteur, Philippe GAMBETTE, sans qui ce projet n'aurait jamais pu aboutir. Pour ses conseils avisés, qui ont fait du générateur un objet complet et performant et pour sa réactivité sans faille, qui nous a permis de perdre le moins de temps possible.

Nous souhaitons également remercier Michel Meynard qui a résolu des problèmes aux proportions qui semblaient insondables.

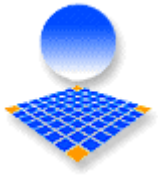
Nous tenons à souligner la gentillesse des administrateurs du site http://www.languefrancaise.net/ , qui ont partagé avec nous le dictionnaire BOB, lexique très complet des mots et expressions d'argot de la langue française. Nous vous invitons d'ailleurs à utiliser ce dictionnaire très complet qui permet de (re)découvrir une foule d'expressions cocasses.

Nous remercions les sites http://www.xp-dev.com/ pour héberger gratuitement notre serveur SVN. ainsi que http://www.dixkey.com/ qui est l'hébergeur gratuit du site.

Nous remercions enfin l'équipe pédagogique du département informatique de l'université de Montpellier II qui nous ont apporté, tout au long de notre formation, les connaissances nécessaires, aussi bien d'un point de vue technique que d'un point de vue organisationnel, pour mener à bien ce projet.

# Table des matières





# 1.Introduction

## 1.1 Généralités

Ce projet est un module de la 3ème année de licence Informatique proposée par l'université de Montpellier II.

Il a pour but la réalisation d'un projet informatique en groupe qui permettra aux étudiants de développer leurs compétences en programmation informatique, mais aussi celle d'organisation - les étudiants devant eux-même réaliser le cahier des charges et le planning prévisionnel – et celle de communication – le projet devant être présenté de manière professionnelle : un rapport complet écrit et une démonstration.

Le projet du générateur équivoques (une url est l'adresse d'une page web, étant l'acronyme de l'expression anglaise *Uniform Resource Locator*, il est possible d'écrire le pluriel avec ou sans -s) est encadré par M. Philippe GAMBETTE qui a aussi le statut de chef de projet. Le groupe d'étudiants qui le réalisera est composé de Nicolas DESBUISSONS, Mikhail DIAKONOV, Éric FONQUERNIE et Stéphane HEUZÉ.

## 1.2 Le sujet

***Contexte et objectif***

Le programme consistera en trois parties principales :

• **concaténation** : concaténer un mot-cible (proposé par l'utilisateur du programme) avec tous les mots possibles du dictionnaire.

• **segmentation** : segmenter la chaîne de caractères concaténée à l'étape précédente en trouvant une segmentation alternative (à partir des mots du dictionnaire).

• **choix de l'utilisateur** : toutes les réponses sont fournies, dans un ordre bien choisi, à l'utilisateur qui sélectionnera manuellement les réponses qui auront le plus de sens.

Si la recherche exhaustive est trop longue dans les deux étapes, elle pourra être raccourcie en identifiant certaines règles de simplification : par exemple, pour l'étape de concaténation, si c'est un nom commun qui est proposé par l'utilisateur, essayer uniquement de le concaténer avec des adjectifs accordés de la même façon.

De même, pour l'étape de segmentation, si de nombreuses réponses existent on pourra identifier des patrons grammaticaux qui ont plus de sens que d'autres (par exemple la séquence [verbe à l'infinitif – adjectif – verbe conjugué au passé simple] risque d'avoir peu de sens), et choisir une stratégie d'exploration qui fournisse le plus vite possible à l'utilisateur des résultats variés.

Une méthode de visualisation permettant à l'utilisateur de bien naviguer visuellement dans

le résultat, pour pouvoir éventuellement compléter manuellement certaines segmentations pendant que l'algorithme poursuit son exploration, sera également fournie.

***Extensions possibles***

L'application web pourra être complétée par une interface d'enregistrement des résultats obtenu, et de vote pour élire les meilleurs résultats.

Le programme pourra être étendu à un dictionnaire phonétique, pour pouvoir générer des holorimes, voire résoudre ou générer des contrepèteries.

Dans la partie segmentation, on pourra envoyer des requêtes automatiques à un moteur de recherche pour compléter le dictionnaire (vérifier si un mot existe en tant que nom propre).

***Étapes***

• Choix du langage et des technologies utiles
• Identification des ressources lexicales nécessaires
• Planification de développement
• Réalisation de composants réutilisables très documentés
• Mise à disposition du programme sur une interface web
• Tests et découverte d'URL équivoques

***Enseignements connexes***

• Langages et automates
• Algorithmes de graphes
• Architecture des applications de la toile

## 1.3 Cahier des charges

### 1.3.1 PARTIE FONCTIONNELLE

### 1.3.2 INTRODUCTION

Le but de notre projet d'Informatique du S6 est de réaliser un logiciel libre à but non-commercial permettant de créer des URL à double sens, c'est à dire pouvant se segmenter en deux suites de mots différentes (par exemple www.penisland.net en anglais ou www.photosdeputes.fr en français).

### 1.3.3 FONCTIONNEMENT GÉNÉRAL

Le programme se présentera sous la forme d'une page web PHP multiplateformes pour l'utilisateur.

Il choisira un thème qui représentera un dictionnaire enregistré sur le serveur. Le dictionnaire de base est le LEFF (http://www.labri.fr/perso/clement/lefff/), dictionnaire de noms communs et noms propres. Il entrera alors un mot. Le générateur, un serveur distant codé en C, lui proposera alors plusieurs URLs équivoques basées sur le mot inséré.

Le programme « regardera » en premier dans une base de données des mots déjà recherchés par d'autres utilisateurs et sauvegardés si un résultat est déjà enregistré. On cherchera ensuite dans le dictionnaire tous les mots qui « matchent » avec le mot de l'utilisateur. C'est-à-dire que pour chaque mot du dictionnaire sélectionné, le programme va « regarder » si les deux, concaténés dans le sens « terme utilisateur » puis « mot du dictionnaire » et « mot du dictionnaire » puis « mot de l'utilisateur », forment deux autres mots.

Par exemple si l'utilisateur entre le mot « kangourou », en faisant défiler le dictionnaire, on trouve le mot « astra » qui est dans le dictionnaire (la marque de Opel, un satellite porte ce nom...) Lorsque l'on concatène les deux termes on obtient « astrakangourou », en découpant différemment ce terme on peut extraire « astrakan » (fourrure) et « gourou » (maître à penser).

On considère que ces deux mots « matchent » car on peut extraire deux autres mots de leur concaténation. On affichera sur la page de l'utilisateur l'association du terme « astra » et « kangourou ».

Ensuite le programme va chercher si la concaténation du mot utilisateur avec un autre terme du dictionnaire peut en créer un troisième.

Par exemple si l'utilisateur a entré « ginseng ». Le programme va chercher dans la liste des mots et concaténer chacun de ceux la avec celui de l'utilisateur pour voir si de l'intérieur de cette chaîne créée on peut extraire un nouveau mot(cf algorithme). Ici on trouvera par exemple « diva » et le programme renverra « di-vagin-seng ». En effet on a pu extraire d'une concaténation le mot «vagin».

L'utilisateur ne verra finalement qu'une page de résultat qui affichera deux parties :
Tous les termes qui « matchaient » avec celui entré, puis toutes les concaténations qui créent un nouveau mot.

L'utilisateur pourra voter (par un clique sur une case à cocher à coté des résultats affichés) pour ceux qu'il préfère. Ce qui permettra de proposer aux prochains utilisateurs cherchant le même termes ceux qui pourraient répondre au mieux à ce qu'il cherche.

Malgré le vote, on sauvegarde toutes les réponses trouvées afin de ne pas à avoir à chercher à nouveau un terme.

## FONCTIONS OBLIGATOIRES

### Traitement du mot entré par l'utilisateur :

– Concaténation avec tous les mots du dictionnaire.
– Segmentation de la chaîne produite en deux partie pour chercher les termes équivoques.
– Affichage de tous les résultats « en vrac ».

### Traitement du dictionnaire :

– Enlever toutes les majuscules du dictionnaire.
– Enlever tous les accents du dictionnaire (pas besoin dans des URLs).
– Trier le dictionnaire par ordre alphabétique.

## FONCTIONS OPTIONNELLES

–Ajouter des dictionnaires à thème.
–Ajouter un dictionnaire spécial pour les termes vulgaires et fripons.
–Traitement en trois mots si on peut extraire un déterminant.
–Enregistrer dans une BDD les résultats déjà obtenus.
–Possibilité pour l'utilisateur de voter pour ses préférés.
–Envoi par mail des résultats à l'utilisateur.

## TRAITEMENT DES DICTIONNAIRES

Le dictionnaire de base est le LEFFF de l'Institut National de Recherche en Informatique et en Automatique , réalisé par Lionel Clément et Benoît Sagot.

Il est organisé de la façon suivante :

–Un fichier texte composé d'un mot par ligne
–La première partie est composée par la ponctuation et les caractères spéciaux.
–La deuxième, des noms propres, triés dans l'ordre ASCII.
– les noms communs ainsi que toutes les formes verbales et adjectivales, triés dans l'ordre ASCII également.

Cette organisation pose plusieurs problèmes.
La ponctuation est inutile, nous la retirerons donc du dictionnaire.
L'ordre ASCII place les termes avec une majuscule initiale en premier, puis les termes non accentués, puis les termes accentués. Comme tous les algorithmes de recherche seront basés sur le principe de la dichotomie, il est impératif que l'ordre alphanumérique soit utilisé. On triera donc tous les dictionnaires afin qu'ils soient triés dans cet ordre. L'algorithme de tri sera basé sur le tri radix.
Travaillant sur des URLs, nous avons décidé de ne pas utiliser d'accents. Même si désormais certains domaines les tolèrent, nous préférons ne pas les utiliser dans un soucis de portabilité. Il existera donc un algorithme pour remplacer tous les termes accentués.

# 2.Organisation du projet

## 2.1 Organisation du travail d'équipe

Pour pouvoir travailler en équipe sans avoir à se retrouver et pour pouvoir échanger les données d'une manière optimisée, nous mettrons en place un serveur de subversion hébergé sur le site http://www.xp-dev.com/. Tous les membres de l'équipe et notre encadrant Philippe GAMBETTE pourront poster et consulter les différentes versions du code, des algorithmes ainsi que les dictionnaires préparés.

La répartition du travaille se fera de la façon suivante : nous créerons ensemble un programme basique opérationnel pour avril, puis nous nous repartirons les fonctions optionnels (cf diagramme de Gantt).

## 2.2 MISE EN PLACE TECHNIQUE DU PROGRAMME

Les normes que nous respecterons pour coder ce projet :

–Le nom des fonctions et des variables seront en « français » de la forme « termeTerme » (le premier terme sans majuscule, le deuxième terme accolé avec la première lettre en majuscule) avec des noms qui explicite le rôle.

- Toutes les boucles auront des accolades.
- On reviendra à la ligne après chaque instruction et chaque accolade. Il y aura trois retours à la ligne entre chaque fonction.
- Il y aura des commentaires complet pour chaque fonction dans les fichiers « .h ».
- La gestion d'erreur ce fera par l'affichage d'une phrase explicite.

L'utilisateur aura accès à une page web codée en PHP. Il n'y aura pas besoin de s'identifier. Il entrera le terme qu'il souhaite dans un formulaire HTML. Le résultat de ce formulaire sera envoyé par un script CGI au programme codé en C++ procédural, qui sera stocké sur un serveur internet. Celui ci renverra à la page PHP un tableau en deux parties qui présenteront toutes les réponses trouvées. L'utilisateur sera invité à cliquer sur les termes qu'il trouve les plus pertinents. On stockera les résultats dans une base de données nommée « URLsEquivoques » stockée sur le même serveur dont la clef sera le terme de l'utilisateur..

Les votes seront stockés dans une tableau nommé « Vote ». Si le terme avait déjà été recherché on rajoute dans le tableau les votes supplémentaires du dernier utilisateur.

L'intégralité des résultats sera dans un tableau nommé « Résultat ».

## 2.3HEBERGEMENT DU SITE ET DE LA BASE DE DONNÉES

Le site web et la base de données, ainsi que le programme seront stockés sur le site dixkey.com. Celui-ci gère tout ce dont nous avons besoin et gratuitement.

# 3.Analyse du projet

Ce projet consiste à extraire des jeux de mots depuis un terme précis. Il faut donc pouvoir définir précisément ce que l'on attend comme type de jeu de mots.

## 3.1 Les types de jeu de mots

### 3.1.1 Jeu de mots équivoque

Le premier type a développer sera sur le modèle de « photosdepute ». C'est-à-dire qu'à un mot de base, on va devoir rajouter un ou plusieurs mots pour créer une petite phrase dont le sens n'est pas univoque suivant comment on l'a lit.

Voici quelques exemples issus du site http://weirdtechnewshub.blogspot.com/

Ceci est un cas du site **www.expertsexchange.com** où des programmeurs peuvent échanger des conseils et des expériences. Les créateurs ont décidé d'appeler leur projet Experts Exchange (échanges d'experts en anglais). Lors de la réservation de domaine, ils ont accolé les deux termes ensembles pour faciliter la mémorisation du nom par les utilisateurs. Malheureusement pour eux, il s'avère que le terme final est « ExpertsExchange ». Ce qui, lu sans la casse, peut se lire « experts exchange ». Mais surtout « expert sex change » (expert en changement de sexe).

On peut voir ainsi le parcours qu'il faut faire pour trouver ce genre de jeu de mot.

La base est que celui-ci est très fréquent dans la langue française. Il est donc impossible de tous les répertorier et ainsi les proposer aux utilisateurs du générateur. Il faut donc que l'utilisateur restreigne le nombre de réponses. Comme il est probable que celui-ci consultera le générateur pour créer un site à thème, pour limiter le nombre de résultat, il faut mettre en place un système qui recherche toutes les urls équivoques sur un mot-clef donné.

Il apparaît que ceci est réalisable par la concaténation d'un mot-clef donc et d'autres mots. Il va falloir que le programme « colle » tous les mots possibles avec le mot-clef. Par chance, il existe déjà des grandes bases de données répertoriant tous les mots de la langue française : les dictionnaires. On utilisera donc un dictionnaire numérique.

Lorsque l'on aura le mot clef, on concaténera celui ci avec tous ceux du dictionnaire. On constate qu'il n'est pas nécessaire que le mot soit le premier ou le dernier de la chaîne crée : D'après l'exemple précédent, si l'utilisateur avait demandé « experts », on voit que le mot a été concaténé par l'arrière. Mais dans le cas de **www.whorepresents.com**, site qui permet de retrouver les agents de stars, d'où le terme « who represents » (qui représente), il s'avère que l'on peut le lire « whore presents » (cadeaux de putain). Ici le mot-clef aurait pu être

« presents » (On tient à préciser que les termes du générateurs seront pour la version de base en français), ce qui montre que le mot-clef peut aussi bien être le premier ou le dernier terme de la chaîne concaténée. Il est important d'offrir le maximum de résultat pertinents.

Une fois que l'on aura le mot-clef concaténé à un autre mot du dictionnaire, il faut « regarder » si l'on peut en extraire deux autres termes différents. On prend la première lettre de le chaîne, on regarde si elle existe. On regarde dans le dictionnaire si le mot composé de la sous-chaîne formée de la chaîne initiale moins le premier caractère existe. Puis on recommence avec les deux premières lettres de la chaîne et la sous-chaîne [2..] et ainsi de suite.

Voici la trace de l'exécution de algorithme si le mot clef est « kangourou » et que le défilement dans le dictionnaire en est au mot « astra » :

```
→ On concatène mot-clef + terme dictionnaire :
                Kangourouastra
  On commence à faire tourner la recherche :
       k angourouastra >>>> mot inconnu
       ka ngourouastra >>>> mot inconnu
       kan gourouastra >>>> mot inconnu
       kang ourouastra >>>> mot inconnu
       kango urouastra >>>> mot inconnu
       kangou rouastra >>>> mot inconnu
       kangour ouastra >>>> mot inconnu
       kangouro uastra >>>> mot inconnu
 kangourou astra >>>> Concaténation initiale
       kangouroua stra >>>> mot inconnu
       kangourouas tra >>>> mot inconnu
       kangourouast ra >>>> mot inconnu
       kangourouastr a >>>> mot inconnu

→ On concatène terme dictionnaire + mot-clef

                astrakangourou
  On commence à  faire tourner la recherche
       a strakangourou >>>> mot inconnu
       as trakangourou >>>> mot inconnu
       ast rakangourou >>>> mot inconnu
       astr akangourou >>>> mot inconnu
       astra kangourou >>>> mot inconnu
       astrak angourou >>>> mot inconnu
       astraka ngourou >>>> mot inconnu
astrakan gourou >>>> URL équivoque valable !
       astrakang ourou >>>> mot inconnu
       astrakango urou >>>> mot inconnu
       astrakangou rou >>>> mot inconnu
       astrakangour ou >>>> mot inconnu
       astrakangouro u >>>> mot inconnu
```

On sauvegarde alors la chaine qui a retournée « URL équivoque valable ». Puis on continu avec les autres mots. L'astrakan est une fourrure de jeune

agneau mort-né à la laine frisée. Le gourou est le chef spirituel d'une secte. « asta » est le nom d'un modèle de voiture. Le kangourou est un marsupial. On a bien, à partir de deux mots (un nom commun et un nom propre) un résultat équivoques : lorsque l'on lit l'url www.astrakangourou.fr on ne peut savoir si l'on parle d'une petite voiture à poche ventrale où d'un leader charismatique amateur de toisons. L'url équivoque est crée.

Un point de réflexion a été amené par les jeu de mots comme « photosdeputes » ou « l'anglophone site » de designers qui le baptisèrent « speed of art ». Il apparaît que de nombreuses possibilités sont offertes si l'on prend en compte les pronoms et les déterminants (de, au, le, cette... (dans notre exemple « of » en anglais). Le traitement de ces derniers se fera sous la forme : nom commun + mot de liaison + nom commun. En effet pour garder un minimum de sens, le résultat doit être grammaticalement juste. Or il y a peu de cas où pronom + déterminant +nom commun ou forme verbale + pronom + nom propre fonctionne. Le but étant de proposer un résultat intéressant à l'utilisateur.

Pour avoir toujours plus de cohérence, il faudra que le mot de liaison et celui qui le suit soient accordés en genre et en nombre. Ainsi il y aura moins de résultats non pertinents.

### 3.1.2 Jeu de mots subliminal

Le deuxième type de jeu de mot que l'on peut traiter consiste, à partir de la concaténation du mot clef et d'un mot du dictionnaire à en extraire un troisième. Ce qui représente une forme plus légère. Une sorte de terme subliminal.

L'exemple peut venir d'un festival de musique classique qui se lance dans les grands opéras et qui s'appelle « colossal opéra ». On constate que le l'algorithme « équivoque » ne renverra pas de résultat puisque il ne gère pas le découpage ternaire et il existe de nombreux cas où les termes qui entoure celui qui retient notre attention ne font pas forcement partie du dictionnaire.

Il faut donc un algorithme qui à partir d'un mot-clef, on trouve tous les termes subliminaux qui pourraient exister. C'est-à-dire pour chaque mot du dictionnaire « collé » au mot-clef (par devant ou par derrière), on va chercher s'il existe une sous-chaîne qui est un mot existant. Pour limiter les recherches, il faudra que ce mot fasse 3 lettres ou plus.
La trace de l'exemple précédent sera :

Avec le mot-clef « opéra » et le terme issu du dictionnaire « colossal »

```
→ On concatène sous la forme mot-clef + terme du dictionnaire
operacolossal

oper ac olossal >>> nom propre trouvé
oper aco lossal >>> mot trouvé sous-chaîne
oper acol ossal >>> pas de sous-chaîne
oper acolo ssal >>> pas de sous-chaîne
oper acolos sal >>> pas de sous-chaîne
```

```
oper acoloss al >>> pas de sous-chaîne
oper acolossa l >>> pas de sous-chaîne
ope raco lossal >>> pas de sous-chaîne
ope racol ossal >>> pas de sous-chaîne
ope racolo ssal >>> pas de sous-chaîne
ope racolos sal >>> pas de sous-chaîne
ope racoloss al >>> pas de sous-chaîne
ope racolossa l >>> pas de sous-chaîne
op eraco lossal >>> pas de sous-chaîne
op eracol ossal >>> pas de sous-chaîne
op eracolo ssal >>> pas de sous-chaîne
op eracolos sal >>> pas de sous-chaîne
op eracoloss al >>> pas de sous-chaîne
op eracolossa l >>> pas de sous-chaîne
o perac olossal >>> pas de sous-chaîne
o peraco lossal >>> pas de sous-chaîne
o peracol ossal >>> pas de sous-chaîne
o peracolo ssal >>> pas de sous-chaîne
o peracolos sal >>> pas de sous-chaîne
o peracoloss al >>> pas de sous-chaîne
o peracolossa l >>> pas de sous-chaîne
On concatène sous la forme terme du dictionnaire + mot-clef

colossalopera
colossa lo pera >>> pas de sous-chaîne
colossa lop era >>> pas de sous-chaîne
colossa lope ra >>> pas de sous-chaîne
colossa loper a >>> pas de sous-chaîne
coloss alo pera >>> pas de sous-chaîne
coloss alop era >>> pas de sous-chaîne
coloss alope ra >>> pas de sous-chaîne
coloss aloper a >>> pas de sous-chaîne
colos salo pera >>> pas de sous-chaîne
colos salop era >>> pas de sous-chaîne
colos salope ra >>> Nom commun trouvé
colos saloper a >>> forme verbale trouvée
colo ssalo pera >>> pas de sous-chaîne
colo ssalop era >>> pas de sous-chaîne
colo ssalope ra >>> pas de sous-chaîne
colo ssaloper a >>> pas de sous-chaîne
col ossalo pera >>> pas de sous-chaîne
col ossalop era >>> pas de sous-chaîne
col ossaloper a >>> pas de sous-chaîne
…
```

On peut ainsi annoncer à l'utilisateur qu'il y a tel et tel mot subliminal.

On constate que cette manière de faire naïve peut être améliorée si l'on regarde dans le dictionnaire si un mot commence par la sous-chaîne sur laquelle. Par exemple « ssalo » n'existe pas, il n'est pas nécessaire d'aller regarder « ssalop » et ainsi de suite.

## 3.2 Problème du nombre de résultats

Le nombre de résultats est un des problèmes phare de ce programme. Il est facile d'imaginer que certains mots vont renvoyer des milliers de résultats. Il faut donc trouver des solutions pour réduire au maximum le nombre de résultats, mais sans se priver de réponses valides.

La première solution trouvée est que c'est l'utilisateur choisira par le biais du site web, le motif grammatical qu'il souhaite. Après avoir entré sont mot il pourra demander la forme nom commun + forme verbale ou adjectif + nom commun... On ne pourra pas imputer au programme des résultats qui n'apparaissent pas ou que le temps de recherche est trop long.

La deuxième solution est de proposer comme une option la gestion des mots de liaison, qui vont avoir tendance à ajouter beaucoup de résultats.

En cherchant des exemples, on s'est aperçu que certaines forme se répétaient pour chaque mot (notamment les pluriels) ou dans certains cas précis.
Exemple :

porcin + oxydable / porc + inoxydable
porcin + faillible / porc + infaillible

Il existe plusieurs cas qu'il faut pouvoir identifier et traiter. Nous indiquerons alors à l'utilisateur une réponse plus générale pour ce genre de cas. « Le terme que vous avez entré est compatible avec tous les mots commençant par « in- ».

Une autre idée, est que sur le long terme, les visiteurs du site chercheront des mots qui ont déjà été demandé. Pour faire gagner du temps sur ceux-la, il suffit d'enregistrer dans une base de donnée le mot-clef ainsi que toutes les réponses. Il faudra ensuite qu'au début de chaque nouvelle recherche on vérifie que le terme n'est pas déjà dans la base, auquel cas on peut l'afficher le récupéré en une fraction de seconde. La recherche consiste juste à sortir les données de la base.

Une solution qui ferait gagner du temps est l'utilisation d'un dictionnaire des mots les plus usités. Ainsi on renvoie en premier lieu les résultats qui correspondent à quelque chose qui parle à l'utilisateur. En effet, le défaut du dictionnaire de base est qu'il propose des termes peu utilisés voire complétement inconnus de la grande majorité des gens.

## 3.3 Les dictionnaires à thèmes

Un aspect qui augmente la durée de vie du programme, son confort d'utilisation et l'optimisation du nombre de résultat est d'ajouter d'autres dictionnaires à celui de base qui est un dictionnaire complet de la langue française. Si l'on peut ajouter des listes de mots à thèmes, les visiteurs seront

plus tentés de l'utiliser : lorsque l'on crée un site internet, il a généralement un thème, et un jeu de mot sur celui ci fera un meilleur effet qu'un plus général.

Il suffira à l'utilisateur de choisir dans quels dictionnaires il souhaite que la recherche se déroule. On gagnera ainsi aussi sur le nombre de résultats, les listes à thèmes contenant bien moins de mots car étant des sous-listes du dictionnaire principal.

Les jeux de mots en général ont d'intérêt qu'il traitent de sujets scabreux. Il faut donc qu'un dictionnaire précis regroupe tous ces termes (grossiers, argots, familiers...)

## 3.4 L'affichage des résultats

L'intérêt d'un tel logiciel est qu'il soit utilisable facilement. Les utilisateurs doivent pouvoir venir avec un thème pour leur futur site, faire une ou deux recherches et avoir un résultat plaisant. Peu de personnes accepteraient de télécharger un programme et de l'installer pour générer une fois une url. Il faut donc une solution simple qui ne demande que peu de temps aux utilisateurs du générateur. La solution la plus évidente est qu'un site internet permette à n'importe qui d'accéder à un formulaire de saisie où l'on rentre le mot que l'on veut voir apparaître dans l'adresse du site et en un minimum de temps recevoir les propositions sur la page et par mail si l'on a pas le temps d'attendre.

Quelqu'un peut aussi visiter le site sans idée précise, il doit pouvoir aussi avoir accès à des propositions qui ont plus aux visiteurs précédents.

Pour savoir ce qui a plu, il faut donc que quand on propose des résultats, le visiteur puisse dire ce qu'il trouve intéressant et pertinent, il doit en un clique le faire savoir. Une solution facile est que devant chaque résultat il y ait une case à cocher. Une fois les choix fait, on clique sur le bouton de soumission et les valeurs sont alors sauvegardées dans une base de données.

On a besoin pour ceci d'un site web qui gère les formulaires, donc du PHP. Il y aura aussi nécessité d'une base de données pour sauvegarder et récupérer les recherches déjà effectuées ainsi que les votes des visiteurs.

Une fois que nous avions toutes ces informations, un plan solide pour savoir comment fonctionnerait notre programme et un cahier des charge précis, nous pouvions commencer à coder.

# 4.Développement des fonctions de recherche des urls

Il a été décidé très tôt que la base de recherche serait le dictionnaire LEFFF (www.labri.fr), un dictionnaire numérisé libre de droit, très complet qui regroupe les noms communs, les noms propres, les pronoms, les déterminants et une grande partie des formes verbales. Il permet aussi de connaître le genre et le nombre de certains termes.

Pour faciliter le traitement du dictionnaire de base et pouvoir appliquer le choix des motifs grammaticaux, il a été fractionné par genre (une fonction a été créé à cette effet).

Pour pouvoir trouver facilement un terme, il fallait une recherche efficace. Comme la liste des mots est triée par ordre alphabétique, la recherche dichotomique est une solution prometteuse. Ceci a malgré tout soulevé un problème : une url est dépourvue d'accent, or la liste issue du LEFFF les comporte, de plus un caractère accentué est classé après son caractère de référence.

Exemple de trie dans le dictionnaire :

rapidement
rieur
réactif

Pour que la recherche renvoie les résultats qui nous intéresse, elle ne doit pas tenir compte de l'accentuation. « réactif » doit se trouver juste après « rapidement ».

Il a donc fallut retirer tous les accents et trier à nouveau la liste (par un tri RADIX). Il a fallut faire la même opération avec les majuscules, pour les mêmes raisons. Après ceci la recherche dichotomique était efficace.

On avait alors à ce moment là, les fonctions suivantes.

- **file_utility.c** est un regroupement en un seul programme de trois fonctions de mise en forme de dictionnaire:

1. une qui remplace les caractères accentués et les majuscules par les caractères correspondants;
2. une qui partitionne le dico en plusieurs fichiers selon la classe grammaticale;
3. une qui trie le dico selon l'ordre de strcmp()

- **searching.c** est le programme recherchant les ambiguïtés en concaténant le mot cible avec tous les mots du dico et affichant toutes les segmentations alternatives.

- **searching.h** contient:

1. la fonction de recherche dichotomique, qui marche à la perfection.
2. une fonction qui détermine la classe grammaticale d'un mot.
3. une fonction qui extrait le mot d'une ligne du dico.

- **word_util.h** contient:

1.une fonction qui extrait une partie d'un mot comprise entre deux indices i et j.

2.une fonction qui concatène deux mots.

3.une fonction qui remplace les caractères accentués et les majuscules d'un mot par les caractères correspondants .

Avec tout ceci la prise en charge d'autres dictionnaires sera facile car :

1. Pour la recherche dichotomique, les seules conditions, c'est un mot par ligne, le mot doit être en début de ligne, et les lignes du dico pas plus longues que 200 caractères (ce qui est facile à modifier).
2. Le fichier doit être classé dans l'ordre de strcmp().
3. La fonction de recherche peut être modifiée pour utiliser un autre ordre sur les mots si nécessaire, il suffit de remplacer strcmp() par ce que l'on veut.

A ce stade, il restait encore quelques problèmes à gérer :

1. Quelques maladresses de gestion de mémoire subsistent encore peut-être;
2. La fonction de tri est toujours en $n^2$
3. Développer une fonction de tri plus performante, en n log(n).
4. Incorporer la gestion des prépositions et déterminants au milieu du mot.
5. Optimiser la performance de l'algo (notamment, en essayant d'éviter les appels à malloc() lorsque c'est possible), et peut-être en l'utilisant sur le dico paritionné plutôt que sur un seul fichier.
6. Trouver / mettre en place d'autres dictionnaires;

Une nouvelle version, optimisée à donc vu le jour. Maintenant, le programme searching.c n'include plus searching.h et word_util.h, qui ne serviront donc plus que pour file_utility ou pour d'autres dictionnaires, et leur contenu, légèrement modifié, a été incorporé dans searching.c. Raison: si l'on appelle une fonction dans un module, celle-ci doit nécessairement effectuer l'opération d'ouverture du fichier et faire des demandes d'allocation de mémoire pour celui-ci avec fopen() ainsi que pour ses variables, avec malloc(). Alors que maintenant, son fichier et ses mallocs sont initialisés en "global", et donc l'allocation de mémoire ne se fait qu'une fois, au lieu de dizaines de milliers de fois, donc ça marche plus rapidement, et il n'y a plus de segfaults à l'exécution

Un gros travail d'optimisation était à ce moment là nécessaire : la concaténation du mot donné avec tous les mots du dictionnaire et la recherche

d'ambiguïtés prenait dans les deux heures. En prenant des dictionnaires partiels (dico des noms communs, dico des adjectifs, etc), on réduisait le temps global proportionnellement à la taille du dico (car tout le fichier est parcouru), donc si l'on remplaçait le lefff global par celui des noms communs, par exemple, qui fait dans les 70k mots, il faut entre 10 et 30 minutes pour faire la recherche. Il semblait difficile d'abaisser ce temps en dessous d'une dizaine de minutes, surtout si on voulait trouver aussi des partitionnements en trois mots ou plus. Donc il fallait trouver une façon adéquate de présenter les résultats...

Après des recherches et une volonté de trouver une solution plus performante, on est arrivé à un nouveau stade :

1. La fonction de tri marche bien. Mikhail a trouvé une fonction intégrée au C (qsort) qui permet de régler le problème.
2. Mise à jour des librairies word_util.h et searching.h; désormais les algos sont plus propres et plus performants et plus adaptables à d'autres dicos.
3. Résolution des problèmes de gestion de mémoire de toutes les fonctions. La concaténation + recherche d'ambiguïtés termine donc sans problème sur le dico de 500.000 mots

L'output des résultats se fait pour le moment dans le terminal pour avoir une visualisation immédiate de l'activité du programme; si besoin on peut les dumper dans un fichier texte.

D'un point de vu performance voici un petit bilan : les premiers tests sur un dico trié et fonctionnel et les algos fonctionnels donnent les résultats suivants: l'algo concaténation + recherche d'ambiguïtés vérifie entre 2 et 4k mots à la minute selon les ordinateurs; ce qui pour la vérification du dico entier donne entre 1h30 et 4h. En revanche, pour les partitions du dico, ce serait moins de temps : en concaténant le mot source seulement avec les adjectifs, par exemple, la recherche prendrait, au total, seulement 25min au grand max.

Au vu de ces résultats, la solution de faire tourner l'algorithme sur tous les mots du dictionnaire et les sauvegarder dans une base de données pour offrir à l'utilisateur une réponse quasiment immédiate prendrait au mieux 60 ans. Cette solution doit probablement être abandonnée

Plus tard dans le développement, le nombre de résultats proposés s'est avéré un problème qu'il fallait résoudre rapidement. Il y avait pour certains termes plusieurs milliers de solutions, il fallait donc poser des contraintes bien plus fortes sans pour autant perdre de résultats intéressants.

1. Tous les mots qui seront traités en trois termes (avec un pronom...) les deux termes encadrant devront être des noms communs.
2. On enlève le dictionnaire des noms propres, sauf si l'utilisateur le spécifie.

3. Quand l'utilisateur utilise le LEFFF, les résultats avec adverbes devront être adaptés : si on extrait 'cette', le nom commun qui ira après devra être féminin singulier.
4. On ne doit pas afficher le pluriel si le singulier a été présenté.
5. Il faut modifier le fichier 'adverbes' pour enlever les personnes.

L'utilisation de la liste de fréquences d'utilisation des mots ( http://sites.univ-provence.fr/~veronis/data/freq-oral.txt ) s'avérait difficilement intéressante, car celle-ci contient seulement 4k mots, contre presque 100 fois plus pour le lefff. Donc la plupart des mots du dico ne figurent pas dans cette liste.

Notre tuteur nous a alors conseillé de s'en servir pour faire ressortir les mots fréquents au début de la liste des résultats, tant pis s'il n'y en a qu'1 ou 2%.

# 5.Développement des dictionnaires à thèmes

Pour pouvoir offrir plus de service aux utilisateurs, l'ajout de dictionnaires à thèmes est une option qui permet une plus grande pertinence dans les résultats proposés et ils augmenteront la durée de vie du programme, puisqu'on pourra en ajouter autant que l'on souhaite.

Pour récupérer les dictionnaires, de longues recherches sur Internet ont été nécessaires. En effet il existe une grande variété de listes de termes sur des sujets précis sur de nombreux site, mais dans la grande majorité des cas, la seule chose à laquelle on peut avoir accès, c'est un formulaire où l'on rentre un terme et le site renvoie la définition du terme. Or il nous fallait récupérer une liste complète rapidement.

Les résultats des recherches ont permis d'obtenir 19 dictionnaires aux termes variés.

Voici la listes des thèmes :

Anatomie, Animaux, Animaux fantastiques, Biologie, Botanique, Culture, Économie, Informatique, Internet, Matériel Informatique, Multimédia, Philosophie, Santé/Beauté, Sciences, Sport et Loisirs, Tourismes et Voyages.

Ils ont été récupérés sur les sites suivants :

http://www.dicodunet.com
http://www.lexilogos.com/
http://fr.wiktionary.org/

Dans certains cas, on pouvait récupérer la liste directement utilisable : une liste des termes où chaque terme est séparé par un caractère spécial.

Dans d'autres cas on n'avait accès qu'à une liste de mots correspondant à une lettre. Il a donc fallut récupérer lettre par lettre dans un fichier.

La méthode pour récupérer ces types de liste est un simple copier coller dans un fichier texte.

Pour quelques dictionnaires, la liste était présentée sans style, c'est à dire que chaque mot était assorti de sa définition mais sans formatage précis, ce qui empêchait un traitement automatisé pour les mettre en forme. Il fallait alors le faire « à la main » : supprimer la définition et mettre un mot par ligne.

Après la récupération il a fallu formater les listes : Suivant le schéma que nous nous étions fixé : un mot par ligne, pas d'accent, pas de majuscule, trié par ordre alphabétique.

On a pu réutiliser les fonctions créé pour le traitement du LEFFF.

Dans les cas où le terme n'était pas un mot unique mais une expression, pour ne pas perdre celle ci, nous avons décidé de remplacer les espaces par un underscore : « _ ».

Ces dictionnaires ne référençant pas les genre et nombre des mots, lorsque l'utilisateur utilisera un dictionnaire à thème, il ne pourra choisir de motif grammatical, sa recherche se fera sur tous les termes du dictionnaire. Ce n'est pas très grave, car la plupart des termes sont des noms communs et les listes ne contiennent que rarement plusieurs milliers de termes.

Nous avions évoquer trouver un dictionnaire des termes scabreux. Les résultats des recherches sur les sites sus-nommés n'offraient que très peu de termes.
Il a donc fallu trouver quelque chose de plus poussé qu'une liste de cent termes.

Nous avons finalement trouvé un site qui proposait un dictionnaire des mots et expressions d'argots : http://www.languefrancaise.net/

Il était très complet et correspondait parfaitement à nos besoins. Comme il ne proposait pas de liste complète directement exploitable, il a fallut se mettre en relation avec le responsable de ce projet. Il nous a répondu en nous proposant gracieusement d'utiliser son dictionnaire qu'il nous a joint.

Celui ci présentait un grand nombre d'expressions mais peu de mots seuls. De plus il était de la forme :
identifiant '« 'terme' »' ' «'VariantesMot' »' ' «'Sens' »' ' «'Expr1' »'

On a donc du créer une fonction qui ne récupère que le terme de chaque ligne, vérifier que ce n'est pas une expression et la sauvegarder dans le fichier final. Enfin on l'a formaté.

Nous avons ainsi pu avoir ce dictionnaire tant convoité.

Pour offrir toujours plus de possibilité à l'utilisateur et faire du générateur un objet puissant de création de jeu de mots, nous avons souhaité ajouter un dictionnaire phonétique. Celui ci permettrait de créer des jeux de mots sur les sons. Un dictionnaire a été trouvé sur le site http://www.lexique.org/telLexique.php. Malheureusement nous n'auront pas le temps de l'inclure dans nos fonctions, les modifications étant trop nombreuses pour pouvoir être traitées à temps.

# 6.Développement de l'interface Web

En partant de l'analyse que nous avions faîtes, réaliser l'interface web semblait être la partie simple du projet :
Un site web en php, un processus de discutions entre le programme et le site réalisée par un script CGI et tous les résultats était facilement sauvegardés dans une base par des requêtes simples SQL.

Il n'en a rien été.

Il s'est avéré après bien des recherches, que l'idée du CGI n'était pas la meilleure. Celui ci ne présente pas la manière optimale pour générer les résultats de manières asynchrone. Ce qui aurait induit que ceux-ci n'auraient été renvoyés qu'a la fin du script. Et au moment où le problème de l'asynchronisme s'est posé, le temps de parution de l'intégralité des réponses avoisinait les deux heures. Le client qui faisait une requête se serait vu immédiatement proposer l'option de revenir dans une paire d'heures. Cette solution étant des pires, car il était peu probable que l'utilisateur ne revienne.

Il a donc fallu trouver une autre solution qui permettait de récupérer les résultats à la voler, afin d'avoir quelque chose à afficher au fur et à mesure de la recherche pour occuper l'utilisateur.

La solution qui a été retenue est le système de socket pour une communication client/serveur. Celui ci permet d'afficher en temps réel les résultats de la recherche.

Une fois que le serveur et le client pouvaient dialoguer, on a pu développer le site en lui-même. Il se présente sous la forme d'une page unique qui contient un formulaire avec une boite texte où l'utilisateur entre son mot. Le mot est traité afin de retirer tout le code malveillant. L'utilisateur valide et le mot est envoyé par la méthode post au programme C qui se charge d'envoyer la requête SQL.

Malheureusement, nous avons découvert que l'éditeur de lien du compilateur C n'appréciait pas particulièrement la bibliothèque mysql.h. Et malgré des recherches et des questions à toutes les personnes susceptibles de pouvoir résoudre cette énigme, nous avons finalement dû abandonner l'idée de passer les requêtes en C, pour aboutir à la solution d'une fonction php.

Une fois que nous avons réussi à faire tomber le temps de recherche, nous avons décidé de laisser tomber la gestion de sessions et les messages, ceux-ci n'avaient plus d'intérêts et surtout parce que le serveur ne peut pas spécifier le message à Mysql vu que ça marche pas.

Donc l'utilisateur arrive à l'accueil (index.php) et choisit les options qu'il veut dans les 3 options proposées (ambigüité inhérente, nom propres et verbe).

Après avoir rentré son mot, l'utilisateur soumet le formulaire (en méthode "post" sur PHP_SELF donc la même page), la socket cliente est crée et se connecte au serveur et y transmet les informations nécessaires (mot et options).

A noter que les fichiers .php ont été enregistré sous format iso-8859-15 (Occidental) afin que l'envoi de données au serveur C s'effectue sans problème vu que les fichier d'extension .c sont enregistrés par défaut sous ce format.

Ce qui a d'ailleurs créé une perte de temps à ce niveau car nous avons cherché l'encodage qu'il fallait en entête des fichiers php alors que c'était le fichier lui même le problème.

De son coté, une fois la requête envoyée, le serveur lance la recherche et rempli un fichier texte contenant tous les résultats que le php va exploiter. (fichier texte : extension du type mot + option1 (0 ou 1) + option2 + option3 + ".txt afin de pouvoir faire la même recherche sur un mot avec différentes options)

Nous avons voulu faire un traitement JavaScript qui afficherait en continu les résultats contenus dans le fichier texte mais seulement l'accès aux fichiers en local depuis JavaScript ne s'effectue qu'avec des fonctions d'ActiveX et sous IE... Donc pas d'affichage en direct des résultats reçus par le fichier texte...

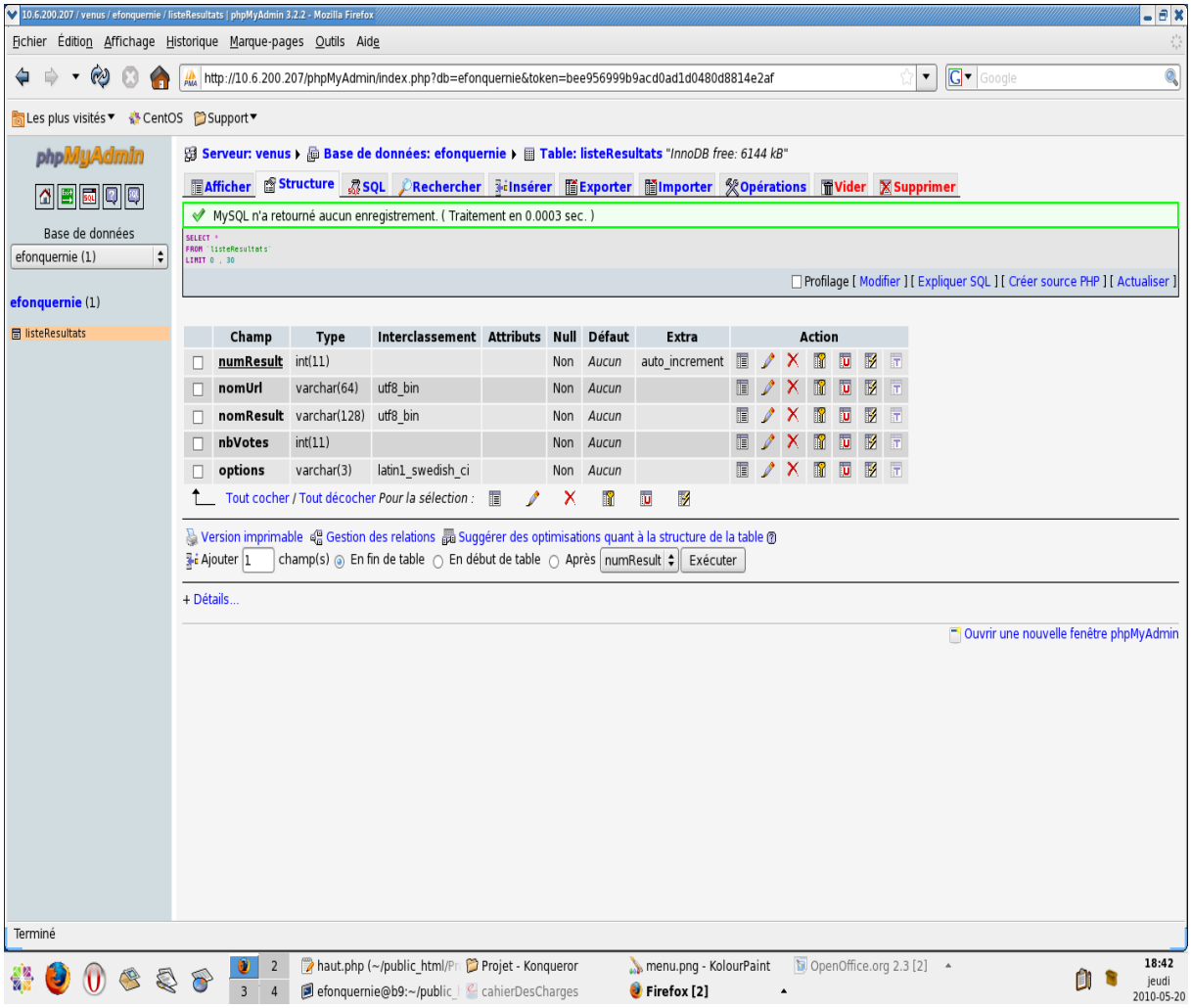
Il a donc fallu trouver une autre approche. C'est pourquoi nous avons mis un time pour dire quand finit la boucle qui est censée afficher les résultats coté client (du style 20-30 secondes) pour que l'utilisateur ait des résultats de suite au lieu de retourner à l'accueil (voir paragraphe suivant) pour les avoir tous.

Pour pouvoir voter il fallait revenir à l'accueil et relancer la recherche car à ce moment là toutes les données du fichier texte étaient transmises à la BDD puis le fichier était détruit. Mais désormais on peut voter à la volée, dès l'affichage du résultat.

Et enfin si quelqu'un cherche un mot déjà enregistré dans la BDD, le résultat apparaît immédiatement sauf qu'il a fallu rajouter un champ par rapport aux options. Une varchar de taille 3, dont le troisième caractère est 1 si l'utilisateur avait inclus dans la recherche les ambiguïtés inhérentes, 0 sinon. 1 comme deuxième caractère si il avait choisit d'afficher les noms propres. Et finalement 1 au premier caractère s'il voulait inclure les verbes.

Le vote s'effectue au clic sur une image par l'intermédiaire d'une fonction JavaScript qui appelle un autre script php mettant à jour Mysql.

Voici comment se présente la base de données.


10.6.200.207 / venus / efonquernie / listeResultats | phpMyAdmin 3.2.2 - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
http://10.6.200.207/phpMyAdmin/index.php?db=efonquernie&token=bee956999b9acd0ad1d0480d8814e2af
Les plus visités CentOS Support
phpMyAdmin
Base de données
efonquernie (1)
efonquernie (1)
listeResultats
Serveur: venus Base de données: efonquernie Table: listeResultats "InnoDB free: 6144 kB"
Afficher Structure SQL Rechercher Insérer Exporter Importer Opérations Vider Supprimer
MySQL n'a retourné aucun enregistrement. ( Traitement en 0.0003 sec. )
SELECT *
FROM `listeResultats`
LIMIT 0 , 30
Profilage [ Modifier ] [ Expliquer SQL ] [ Créer source PHP ] [ Actualiser ]
Champ Type Interclassement Attributs Null Défaut Extra Action
numResult int(11) Non Aucun auto_increment
nomUrl varchar(64) utf8_bin Non Aucun
nomResult varchar(128) utf8_bin Non Aucun
nbVotes int(11) Non Aucun
options varchar(3) latin1_swedish_ci Non Aucun
Tout cocher / Tout décocher Pour la sélection :
Version imprimable Gestion des relations Suggérer des optimisations quant à la structure de la table
Ajouter 1 champ(s) En fin de table En début de table Après numResult Exécuter
+ Détails...
Ouvrir une nouvelle fenêtre phpMyAdmin
Terminé
haut.php (~/public_html/Pr Projet - Konqueror menu.png - KolourPaint OpenOffice.org 2.3 [2]
efonquernie@b9:~/public_ cahierDesCharges Firefox [2]
18:42
jeudi
2010-05-20

# 7. Manuel d'utilisation

L'utilisation de ce programme est fort simple.

L'utilisateur va sur le site.

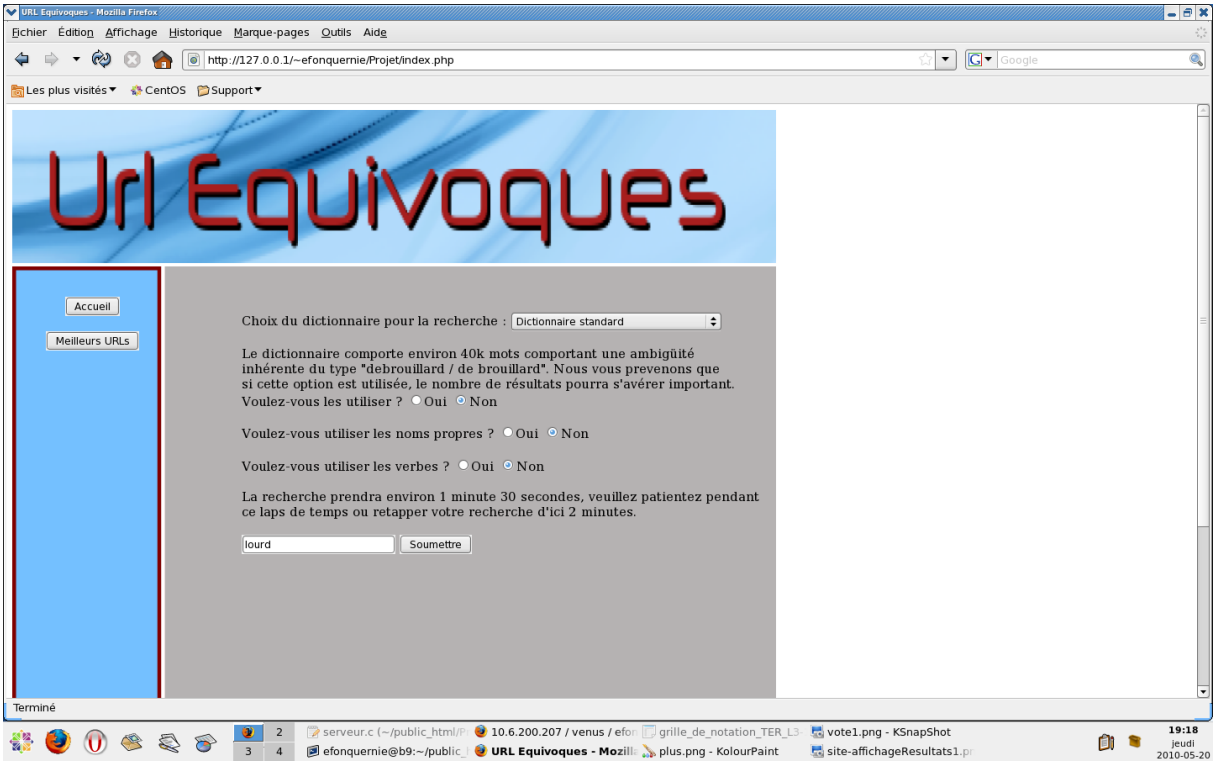
Ici il trouve un formulaire avec une zone texte où il peut choisir son mot.
Il va pouvoir alors choisir dans une liste de tous les dictionnaires à thème celui qu'il souhaite utiliser. Le dictionnaire standard contient un grand nombre de mots et permet la gestion des genre et nombre.
Les autres contiennent moins de mots et ne gèrent par les genres. Mais les résultats seront sans doute plus pertinents si l'on veut que l'url générée ait un sens lié au thème du site qui accueillera cette url.
Le dictionnaire d'argot contient un grand nombre d'entrées des termes argotiques, mais aussi familier et vulgaires. Ainsi qu'un grand nombre d'expressions.

Trois options lui sont proposées :

- Inclure dans la recherches les ambiguïtés inhérentes, c'est à dire tous les termes qui commence ou finissent par un mot de liaison. Par exemple le verbe « débiter » à la première personne du singulier renverra toujours un résultat quel que soit le nom commun entré par l'utilisateur.
- Inclure ou non les noms propres.
- Inclure ou non toutes les formes verbales.

Ces trois options représentes un grand nombre de résultats supplémentaires.

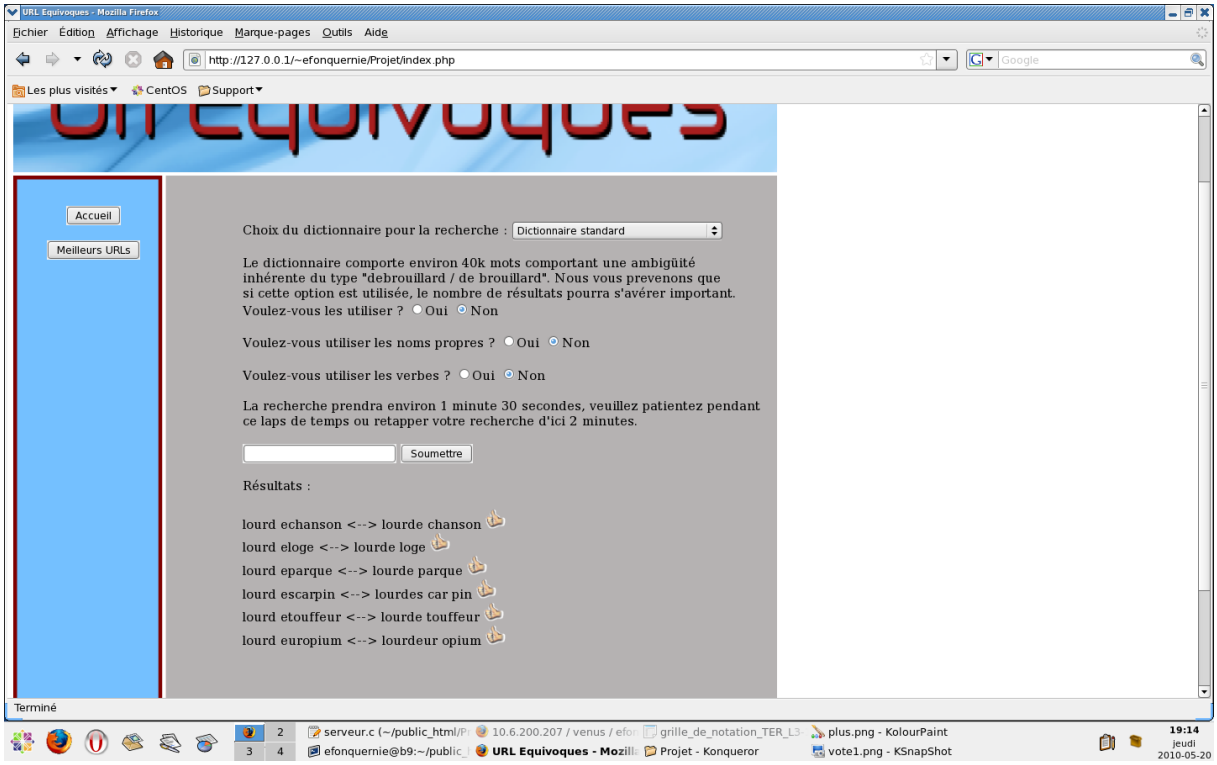
Voici comment se présente le site



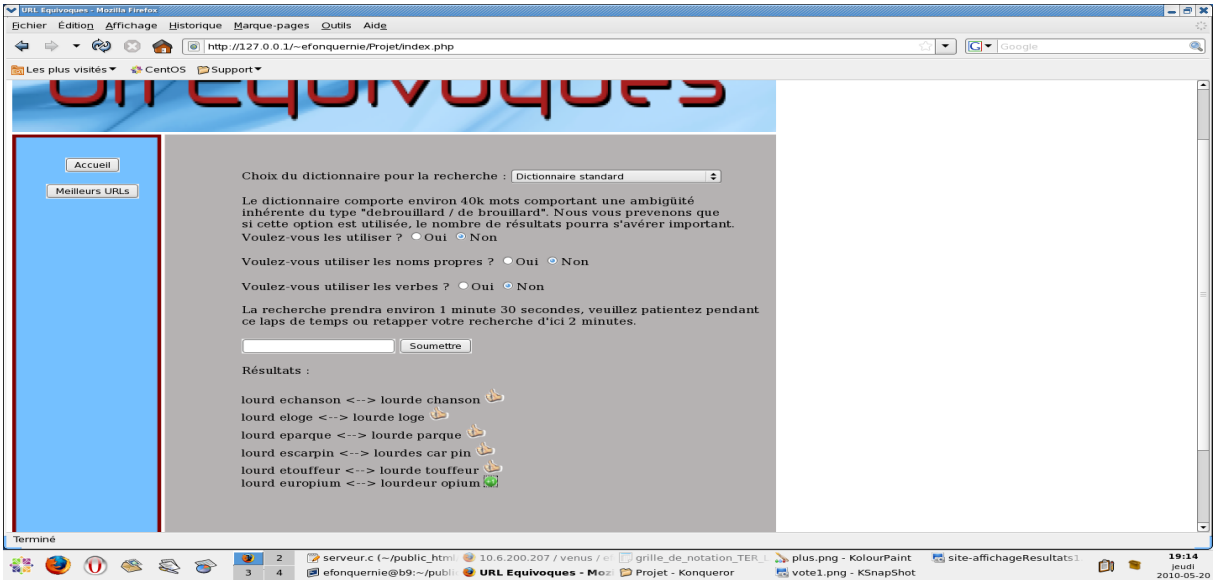
L'utilisateur n'a plus qu'a cliquer sur « soumettre ».

Si le serveur est bien lancé, il testera si le mot est dans le dictionnaire et le cas échéant à définir son genre et son nombre, pour affiner les recherches. Sinon il testera tous les mots du dictionnaire.

La recherche s'effectuera suivant les critères sélectionnés pendant 1 minute 30. Les résultats s'afficheront à la volée

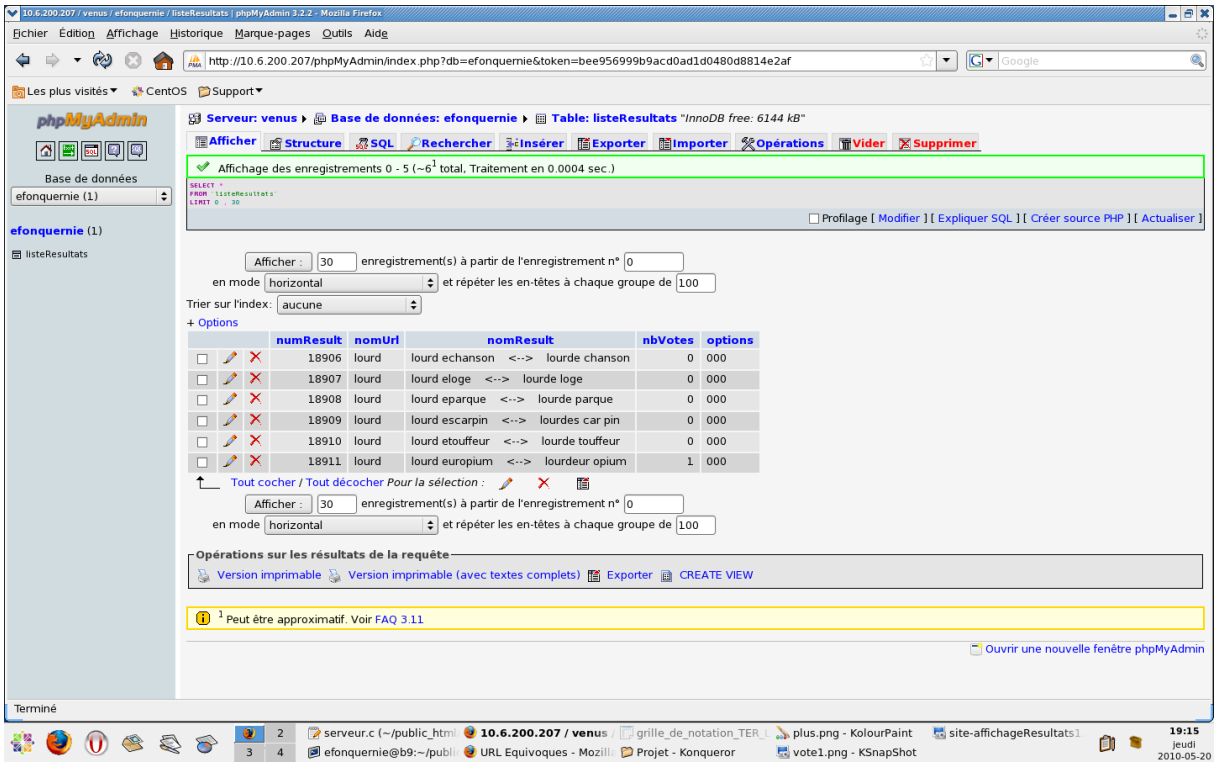



L'utilisateur sélectionnera alors, s'il le souhaite sur les urls qu'il préfère. Si il ne l'a pas déjà fait, il n'aura qu'a cliquer sur le petit pouce qui est à coté de l'url générée, ce qui le transformera en un point vert qui signifie que le vote a bien été prit en compte



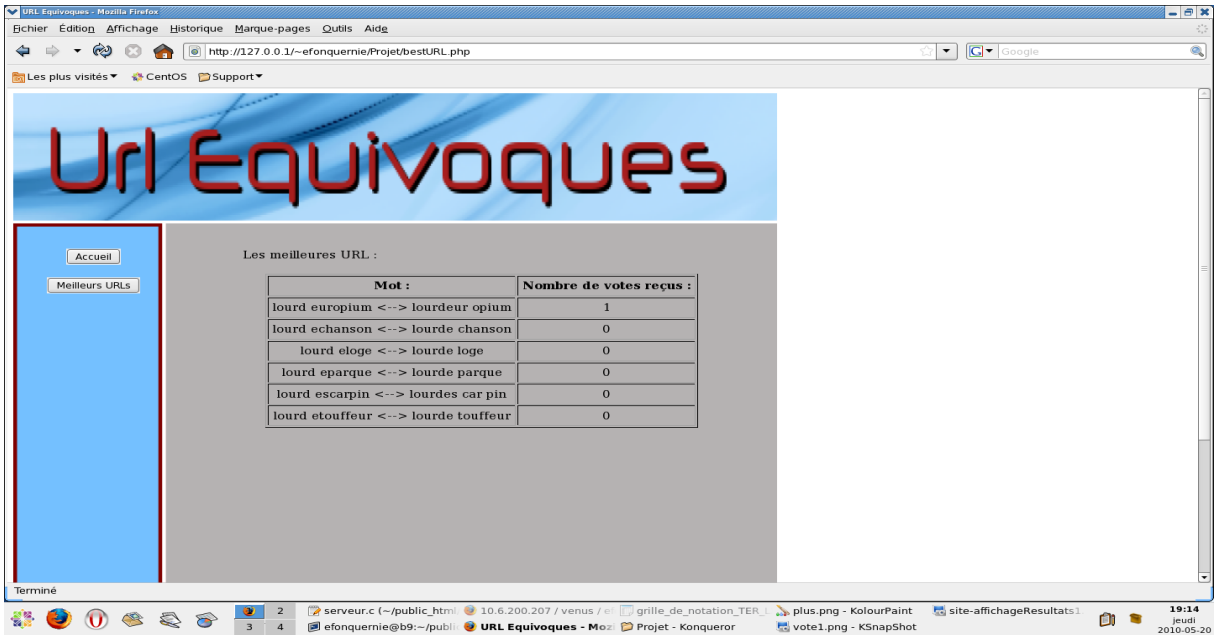
Tous les résultats et les votes seront sauvegardé dans la base de données.

S'il arrivait que l'utilisateur quitte le site durant la recherche des mots, celle-ci continuerait jusqu'à la fin, et quand une nouvelle requête sur ce mot serait réalisée, l'affichage se ferait immédiatement, les réponses étant intégralement sauvegardées dans la base.




On voit ici les données sauvegardées dans la base.

Sur la page principale l'utilisateur aura aussi le choix d'afficher les 30 urls ayant reçu les meilleurs votes.



Nous avons donc ici un programme extrêmement facile à utiliser, produisant des résultats exploitables avec un coup mémoire très réduit et une rapidité exceptionnelle.

Le code de celui ci a été prévu pour une optimisation et des modifications faciles à mettre en œuvre.

# 8.Perspectives et conclusions

## 8.1 Perspectives

Ce projet peut à présent générer efficacement n'importe quelle url équivoque. L'utilisateur peut l'utiliser facilement, choisir des restrictions intéressantes (le choix d'un thème particulier rend l'exercice ludique pour celui qui peut le faire, tester les réponses proposées...).

Continuer dans cette veine ludique peut être le meilleur moyen pour, plus qu'augmenter la durée de vie du logiciel, en faire un structure vraiment puissante dans le domaine du jeu de mots.

Pour la durée de vie, il suffirait que l'utilisateur puisse entrer son propre dictionnaire. Prenons l'exemple des sites de fans (d'un film, d'un manga, d'un jeu) où souvent l'on voit des fans faire leur propre blog, leurs histoires... S'ils pouvaient ajouter le dictionnaire qui regroupe les termes de leur passions, le générateur d'urls deviendrait un générateur de titres plus général. Titre de livres, de films, de rubriques... Le jeu de mots spécialisé créerait un attrait non négligeable.

Pour mettre en œuvre une telle fonctionnalité, il faut une fonction qui teste si le dictionnaire proposé est bien valide : un mot par ligne, que des caractères alphabétiques... Il faut faire très attention à la faille de sécurité que cela représente : travailler directement sur du texte de l'utilisateur est toujours quelque chose de risqué, même si l'on enlève les caractères spéciaux. Il faudrait indiquer pour chaque erreur à l'utilisateur de la corriger et de soumettre à nouveau sa liste. Ce qui est une opération qui peut s'avérer répétitive et contraignante pour celui ci.

Une fois la liste acceptée, on a plus qu'à lui appliquer le formatage que nous avons déjà cité. Enfin il faudrait ajouter la possibilité aux autres utilisateurs de sélectionner cette nouvelle liste.

Ce traitement permettrait d'ajouter des dictionnaires d'autres langues.

Le générateur pourrait aussi être grandement optimisé si l'on pousse le développement sur le dictionnaire phonétique. Car avec celui ci, nous pourrions générer tout aussi bien des contrepèteries ("*J'ai le sang qui bout.*" devient "*J'ai le bout qui sent.*" ) que des calembours ("*Demandez nos exquis mots !*" ) ou encore des poèmes holorimes ("*Par les bois du Djinn, où s'entasse de l'effroi, parle et Bois du gin ! ... ou cent tasses de lait froid.*" )...

Toutes ces formes de jeux sur les sons peuvent être automatisées.

Prenons l'exemple de la contrepèterie. Si l'on partait d'un mot, on pourrait chercher tous les termes qui contiennent ses phonèmes et tester si des combinaisons sont possibles.

Bien évidement une telle recherche prendrait beaucoup de temps, vu le nombre de possibilités. Il faudrait veiller aussi au respect de l'ordre grammatical dans la phrase... Sans être facile, il est tout à fait faisable de transformer le

générateur en une plateforme qui en partant de peu (un terme) arrive à trouver des jeux de mots avancés.

## 8.2 Conclusions

### 8.2.1 Fonctionnement de l'application

De nombreuses difficultés ont été rencontrées au cours de ce projet. Arriver à réduire le temps d'attente de l'utilisateur de plusieurs heures à quelques minutes a été un réel défi. Puis il a fallu trouver des moyens pour que le nombre de résultats ne soit pas trop grand pour ne pas rebuter l'utilisateur, mais sans pour autant perdre de résultats pertinents.

Un autre problème est venu des échanges entre le programme et le site web. L'envoie de requêtes SQL en C s'est avérée si problématique que nous avons finalement dû y renoncer.

Il a été au final difficile de travailler assez de concert pour pouvoir ajouter la fonction de recherche des jeu de mots subliminaux.

Créer un site avec une interface plus professionnelle aurait été un atout de plus.

Mais au final, nous avons un programme qui fonctionne bien et rapidement. L'interface pour l'utilisateur est facile à prendre en main et offre de bons résultats.

Le générateur est désormais un logiciel stable et facile à optimiser. Le choix des langages se sont avérés bon. Il est tout de même regrettable que la mise en ligne n'ait pu se faire suite à des difficultés d'organisations.

Le langage C a permis de gagner beaucoup de temps sur les recherches grâce à une gestion optimale des pointeurs. Il a en revanche posé des problèmes pour les requêtes SQL mais ce problème a pu largement être rattrapé du coté PHP.

Ce dernier a d'ailleurs permis d'obtenir un site complet, qui répond à toute nos attentes, tout aussi bien pour la communication avec le serveur que pour retrouver les données dans notre base.

Lors de l'analyse, nous avions cerné la majorité des problèmes survenus, mais nous les avions sous-estimés, la preuve étant qu'il a fallut par plusieurs fois optimiser la recherche pour obtenir des urls dans un temps convenable.

Mais au bout du compte le projet a été mené sérieusement dès le début, ce qui a permis d'offrir le résultat que nous avons aujourd'hui

### 8.2.2 Fonctionnement du groupe de travail

Il n'a pas toujours été facile de se réunir pour aborder les problèmes de vive voix. Mais grâce à l'envoie de mail et à la publication des problèmes sur le forum dédié au projet couplé à la réactivité de chaque membre, les différentes parties du projet n'ont jamais stagnées à cause d'un problème insoluble.

Comme l'analyse du projet à été bien menée, nous avons pu repartir les différentes tâches en fonction des qualités et des goûts de chacun. C'est sans doute ce qui a fait que le projet a pu être fini dans les temps, chacun étant motivé par la partie qu'il avait à réaliser. Et même si le diagramme de Gantt n'a pas été suivi au pied de la lettre, il y a eu une réelle continuité dans l'avancement des tâches à accomplir.

C'est parce que chacun s'est investit dans le projet tout en étant prêt à aider les autres, à y investir le temps nécessaire et tenir informer les autres membres que le générateur d'urls équivoques est désormais un logiciel à part entière aussi efficace et facilement optimisable.

Nous espérons que vous prendrez plaisir à tester toutes les possibilités qu'offre ce générateur et qui sait peut être découvrir des jeux de mots jusque la restés inconnus de tous.

# 9.Annexe A : documents d'analyse

Voici le suivit au jour le jour de l'avancement du projet.

## 9.1Réunion du 25 février 2010

Réunion avec l'encadrant.

**Réflexions :**

- Le dictionnaire de travail sera le LEFFF (www.labri.fr)
- Il faudrait que le programme soit multi plateformes.
- l'utilisateur peut proposer un motif (nom/adverbe, verbe/adjectif...).

**Décisions :**

- Le programme renvoie tous les découpages possible du Mot Utilisateur.
- En avril le programme devra tourner basiquement
- Le cahier des charges :
- Il devra présenter : le choix du langage; comment va se faire la communication avec l'interface graphique;
- un calendrier des taches et la répartition des charges (digramme de Gantt).
- le langage sera C++ pour le programme, PHP pour l'interface web.

**A faire :**

- Choisir quel sera le moyen de communication programme/interface
- Voir si l'utilisateur peut choisir un dictionnaire à thèmes.

## Réunion du 10 Mars 2010

**Fait :**

- Le dictionnaire LEFFF a été fractionné en genres (une fonction a été créée).
- la fonction qui découpe la concaténation de deux mots en deux autres mots et affiche si les mots nouvellement crées existent.
- la recherche dichotomique sur le dictionnaire fonctionne.

**Réflexions sur le cahier des charges :**

Il faut enlever la casse du dictionnaire ainsi que les accents pour pouvoir trier dans l'ordre alphanumérique (a cause de la recherche dichotomique)
afficher les langages choisis

les fonctions obligatoires :

1. concatène(string a1, string a2)
2. segmentation (string a1a2)
3. (explications des fonctions dans le paragraphe "Fait" de ce jour.
4. Une fonction qui affiche tous les résultats en vrac.
5. Fonctions de nettoyage du dictionnaire

les fonctions optionnelles :

1. travail sur les déterminants et préposition.

2. faire un choix du thème du dico
3. ajouter un dico ‘sale’ pour les blagues 'undergrounds'
4. enregistrer le résultat
5. vote de l’utilisateur des meilleurs réponses.

**Décisions :**

Il faut une base de données pour enregistrer les résultats et les votes.
Principe de fonctionnement de l’application :
l’utilisateur aura accès a une page web qui envoie une requête prédéfinie avec le mot demandé, a un serveur qui tourne. le serveur renvoie les réponses qui seront affichées sur la page de l’utilisateur. Il sélectionnera celles qu’il jugera les plus pertinentes. On sauvegarde tous les résultats dans la base de données et on demande les notes.

**A Faire :**

- Le cahier des charges
- les fonctions du dictionnaire.

## Réunion du 11 mars 2010

**Fait :**

- Fonction qui trie le dictionnaire (fichier texte) dans l’ordre alphanumérique (Tri bulle complexité $n^2$)
- Fonction qui trouve le genre des mots dans le dictionnaire

**Réflexions :**

- Concaténations des mots suivant un patron.
- Écrire les algorithmes et les schémas de segmentation.
- Se renseigner sur sur le distribution et les termes du contrat d’utilisation du LEFFF.
- Trouver où stocker le programme sur le net.
- Faire le CDC en trois parties :

# Fonctionnelle : A peu près ce qu’on a là, sans rien de technique. Parler des problèmes d’accents. # Technique : Algos, pseudo-code, règles de normage (ang/fr), décrire le format des fichiers. # Prévisions : Gantt + répartition. * Mentionner le LEFFF.

- Mettre en place un tri RADIX.

**Décisions :**

- Les normes de nommage seront les mêmes que dans le cahier des charges présenté sur le site de Meynard.

**A FAIRE :**

- La première partie du travail pour avoir une base fonctionnelle.
- On se partagera ensuite le travail sur les options.
- Envoyer un mail à M. Gambette avec le cahier des charges.

## Mail du 13 Mars

- Un lien vers un tuto pour la communication C++/PHP via des sockets.

http://www.forum.moteurprog.com/reseaux/forum-msg-18019-1.htm

## Mail du 16 Mars

**Fait :**

- file_utility.c est un regroupement en un seul programme de trois fonctions de mise en forme de dictionnaire:

1. une qui remplace les caractères accentués et les majuscules par les caractères correspondants;
2. une qui partitionne le dico en plusieurs fichiers selon la classe grammaticale;
3. une qui trie le dico selon l'ordre de strcmp() (pas encore très au point, celle-là, car en $n^2$).

- searching.c est le programme recherchant les ambiguïtés en concaténant le mot cible avec tous les mots du dico et affichant toutes les segmentations alternatives.

- searching.h contient:

- la fonction de recherche dichotomique, qui marche à la perfection;
- une fonction qui détermine la classe grammaticale d'un mot;
- une fonction qui extrait le mot d'une ligne du dico.

- word_util.h contient:

- une fonction qui extrait une partie d'un mot comprise entre deux indices i et j;
- une fonction qui concatène deux mots;
- une fonction qui remplace les caractères accentués et les majuscules d'un mot par les caractères correspondants

Au niveau algorithmique à proprement parler, tous les objectifs de base sont donc remplis.

**Réflexions :**

- Le dictionnaire à utiliser est, pour le moment, le lefff avec les accents, car malgré les inconvénients, la recherche dichotomique fonctionne sur celui-ci.
- Il faut supprimer du fichier dico toutes les lignes au début et à la fin qui ne font pas partie de la liste des mots à proprement parler (CGU, commentaires).
- l'encodage du input doit se faire en ISO 8859-15, du moins si l'on veut que les caractères accentués soient gérés.

*Utilisation d'autres dictionnaires:

1. A priori, ça devrait être simple, car pour la recherche dichotomique, les seules conditions, c'est un mot par ligne, mot en début de ligne, et lignes du dico pas plus longues que 200 caractères (facile à modifier)
2. Le fichier doit être classé dans l'ordre de strcmp(), d'où l'utilité d'une fonction de tri performante (à venir)
3. La fonction de recherche peut être modifiée pour utiliser un autre ordre sur les mots si nécessaire, il suffit de remplacer strcmp() par ce que l'on veut
4. Les fonctions déterminant la classe grammaticale ou extrayant le mot de la ligne sont très facilement adaptables à d'autres structures de données

**A Faire :**

1. Quelques maladresses de gestion de mémoire subsistent encore peut-être;
2. Le dictionnaire n'a pas encore été trié;
3. La fonction de tri est toujours en $n^2$
4. développer une fonction de tri plus performante, en n log(n) (c'est l'optimum pour un algo de tri);

5. incorporer la gestion des prépositions et déterminants au milieu du mot (facile, rajouter quelques lignes de code suffira);
6. optimiser la performance de l'algo (notamment, en essayant d'éviter les appels à malloc() lorsque c'est possible), et peut-être en l'utilisant sur le dico paritionné plutôt que sur un seul fichier (à voir)
7. Tout ce qui est interface web ainsi que stockage et affichage des résultats (je crois que Eric a commencé à faire un truc, mais il en dira plus par lui-même)
8. Communication C – PHP (je crois que Eric a aussi des idées dans ce domaine?)
9. Trouver / mettre en place d'autres dictionnaires;

## Mail du 18 Mars

**Fait :**

- Une nouvelle version, optimisée, de la fonction de recherche. Maintenant, le programme searching.c n'include plus searching.h et word_util.h, qui ne serviront donc plus que pour file_utility ou pour d'autres dictionnaires, et leur contenu, légèrement modifié, a été incorporé dans searching.c. Raison: si l'on appelle une fonction dans un module, celle-ci doit nécessairement effectuer l'opération d'ouverture du fichier et faire des demandes d'allocation de mémoire pour celui-ci avec fopen() ainsi que pour ses variables, avec malloc(). Alors que maintenant, son fichier et ses mallocs sont initialisés en "global", et donc l'allocation de mémoire ne se fait qu'une fois, au lieu de dizaines de milliers de fois, donc ça marche plus rapidement, et il n'y a plus de segfaults à l'exécution (merci Meynard pour le tuyau =) )

**Réflexions :**

- la concaténation du mot donné avec tous les mots du dictionnaire et la recherche d'ambiguïtés prendrait, à vue d'nez, dans les deux heures; l'algo est déjà assez minimaliste, donc on ne pourra pas faire beaucoup mieux;
- en prenant des dictionnaires partiels (dico des noms communs, dico des adjectifs, etc), on réduit le temps global proportionnellement à la taille du dico (car tout le fichier est parcouru), donc si l'on remplace le lefff global par celui des noms communs, par exemple, qui fait dans les 70k mots, eh bien, il faudra quand même 10-20-30 mins pour faire la recherche, et on ne pourra jamais descendre, je pense, en dessous d'une dizaine de minutes, surtout si on veut trouver aussi des partitionnements en trois mots ou plus. Donc il faut penser à une façon adéquate de présenter les résultats...

**A Faire :**

- gérer l'histoire des déterminants au milieu de mots (à venir), et implémenter un tri efficace.

## Mail du 22 Mars

**Fait :**

- La fonction de tri marche nickel; Mikhail à trouver une fonction intégrée au C (qsort) qui permet de régler le problème.
- Mise à jour des librairies word_util.h et searching.h; désormais les algos sont plus propres et plus performants et plus adaptables à d'autres dicos (en pièce jointe).
- Résolution des problèmes de gestion de mémoire de toutes les fonctions; la concaténation + recherche d'ambiguïtés termine donc sans problèmes sur le dico de 500k mots (en pièce jointe).

- L'output des résultats se fait pour le moment dans le terminal pour avoir une visualisation immédiate de l'activité du programme; si besoin on peut les dumper dans un fichier texte.

**Réflexions :**

- Les premiers tests sur un dico trié et fonctionnel et les algos fonctionnels donnent les résultats suivants: l'algo concaténation + recherche d'ambiguïtés vérifie entre 2 et 4k mots à la minute selon les ordinateurs; ce qui pour la vérification du dico entier donne entre 1h30 et 4h. En revanche, pour les partitions du dico, ce serait moins de temps : en concaténant le mot source seulement avec les adjectifs, par exemple, la recherche prendrait, au total, seulement 25min au grand max. De même, les résultats sont parfois abondants: par exemple, en prenant comme mot source "porcin", on a tous les résultats qui correspondent aux mots qui sont négation:

porcin + oxydable / porc + inoxydable
porcin + faillible / porc + infaillible
etc etc.

Ce qui fait que qu'il n'est pas forcément pertinent de les afficher sur une page web, qui sera bien souvent remplie par des résultats qui se ressemblent en assez grand nombre, et ferait l'utilisateur attendre trop de temps devant son ordinateur.

- L'identification de patrons grammaticaux est-elle vraiment judicieuse ? car pour chaque patron grammatical que l'on voudrait rejeter, car grammaticalement incorrect, on peut trouver un exemple de résultat "juteux" qu'il serait dommage de jeter.
- Si l'on part sur la base de 1h30 pour un mot (Un ordinateur puissant ça se trouve), Avec un dico de 500.000 mots. Si l'on veut préalablement entrer tous les mots dans la base de données avant la mise en service du site, ça nous donne un temps de traitement de 85 ans... Mais ca part vite, si on passe à une heure par mot ça ne fait plus que 57 ans. Cette solution doit probablement être abandonnée.
- Proposer, en bonus, des jeux à base de mots... Voire faire carrément concurrence au site de Lafourcade =)

**Proposition de fonctionnement général :**

1. L'utilisateur entre son mot, puis choisit sur un menu déroulant avec quelle classe grammaticale il veut le concaténer (tous, verbe, adjectif, nom commun, etc).
2. Sur sélection, on vérifie dans la base de données si le mot proposé par l'utilisateur a déjà été analysé: si oui, l'utilisateur en est prévenu et on propose de visualiser les résultats; si non, une fois la recherche terminée, les résultats seront entrés dans la base de données: ainsi, au fur et à mesure, on aura de plus en plus de résultats immédiatement disponibles.
3. Dans le menu déroulant, on peut également préciser le temps de recherche estimé pour chaque choix afin de prévenir l'utilisateur.
4. L'utilisateur, plutôt que d'attendre entre 20 min et 4h devant une page web, recevra les résultats de la recherche par mail sous forme de fichier texte (ou lien vers la page, ou simple notification que ses résultats ont été entrés dans la BDD et qu'il peut repasser sur le site pour les consulter).
5. Dans ces conditions, il peut être judicieux de remplacer le vote en direct pour les meilleurs résultats par un classement des mots "les plus populaires": à chaque recherche d'un mot en particulier, on incrémentera le compteur pour ce mot. Si l'on opte cependant pour notifier l'utilisateur par mail que sa recherche est terminée et qu'il peut passer sur le site voir les résultats, on peut quand même ajouter un système de notation des résultats... Ou utiliser les deux systèmes.

## Réunion du vendredi 16 avril 2010

**Réflexions :**

1. Le nombre de résultats affichés est trop grand, il va falloir poser des contraintes plus fortes sans pou autant perdre de résultats intéressants.
2. Il y a un problème avec les ordinateurs de la fac pour importer mysql.h

**Décisions :**

1. Tous les mots qui seront traités en trois termes (avec un pronom...) les deux termes encadrant devront être des noms communs.
2. On enlève le dictionnaire des noms propres, sauf si l'utilisateur le spécifie.
3. Quand l'utilisateur utilise le LEFFF, les résultats avec adverbes devront être adaptés : si on extrait 'cette', le nom commun qui ira après devra être féminin singulier.
4. On ne doit pas afficher le pluriel si le singulier a été présenté.
5. Il faut modifier le fichier 'adverbes' pour enlever les personnes.
6. Tout le code devra être commenté

## Le 19 avril 2010.

- Des améliorations au niveau du code qui s'en trouve allégé.
- le fichier dico ne doit plus commencer par une ligne vide.
- On facilement les catégories grammaticales des mots du dico, c'est-à-dire ce que l'on lit sur les lignes après @ et avant la fin, comme par exemple @ms, ou @fp.

- Réflexions :*

1. L'utilisation de la liste de fréquences d'utilisation des mots ( http://sites.univ-provence.fr/~veronis/data/freq-oral.txt ) s'avère difficilement intéressante, car celle-ci contient seulement 4k mots, contre presque 100 fois plus pour le lefff. Donc la plupart des mots du dico ne figurent pas dans cette liste.
2. RÉPONSE : "Ce n'est pas vraiment un problème, l'idée est de faire particulièrement ressortir les mots fréquents au début de la liste des résultats, tant pis s'il n'y en a qu'1 ou 2%. Pour les mots qui ne sont pas dans la liste, mettez-leur un score de 0."

- A Faire :*

- "Je pense que je vais, dans un premier temps, filtrer de façon grossière grâce aux classes grammaticales ce qui permettra de ne pas coller un adjectif avec les verbes, ou un verbe + mot de liaison + un nom commun. Ensuite, on peut atteindre un degré de correction grammaticale presque parfait grâce aux indications du dictionnaire, mais là ça va demander un sacré travail de scribe, car il y a énormément de cas à traiter."

## Le 20 avril 2010.

- Ajout de 19 dictionnaires à thèmes.
- Prise de contact pour en récupéré un complet d'argot sur le site http://www.languefrancaise.net/

# 10. Annexe B : Listings identifiés et commentés

```
#include "searching.h"
#include "word_util.h"


#define TAILLE_MAX_LIGNE 200




/*
Fonction qui prend en argument un fichier, et en crée une copie nommée fichierstd.txt, avec les caractères accentués et les majuscules remplacés par le caractère correspondant:
é --> e,
ô --> o,
A --> a,
etc.
*/

void standardizeFile(char filename[])
{
    char ligne_lue[TAILLE_MAX_LIGNE];
    FILE* fichier = NULL;
    FILE* output_fichier = NULL;
    char* resultat = (char*)calloc(TAILLE_MAX_LIGNE, sizeof(char));
    char* resultat_st_mot = (char*)calloc(TAILLE_MAX_LIGNE, sizeof(char));

    strcpy(resultat, filename);
    strcat(resultat, "std.txt");


    fichier = fopen(filename, "r");
    output_fichier = fopen(resultat, "w");

    while(fgets(ligne_lue, TAILLE_MAX_LIGNE, fichier) != NULL)
    {
        fprintf(output_fichier, standardizeMot(ligne_lue, resultat_st_mot));
        printf("processing line.... %s", ligne_lue);
    }
    printf("standardisation terminée :'-)\n");

    free(resultat);
    free(resultat_st_mot);
    fclose(fichier);
    fclose(output_fichier);
```

```
}
```

```
/*
Fonction qui classe le fichier donné en argument par ordre de strcmp().
Tri utilisé: tri rapide (quicksort) complexité O(n²) dans le pire des cas, O(nlogn) en moyenne. ( http://fr.wikipedia.org/wiki/Tri_rapide )
Exemple de liste à trier:
zorro
abba
raton-laveur
term_signal
tèrm_signal
123
ôter
afterchezlulu
dix-huit
vingt-quatre
*/

int cmp(const void* mot1, const void* mot2)
{
    return strcmp(*(const char**)mot1, *(const char**)mot2);
}


void sortFile(char chemin[])
{
    int i = 0;
    long int nombre_lignes = 0;
    char ligne_lue[TAILLE_MAX_LIGNE];
    char c[TAILLE_MAX_LIGNE];
    char* resultat = (char*)malloc(strlen(chemin) + strlen("sorted.txt") +1);
    FILE* fichier = NULL;
    FILE* output_fichier = NULL;
    char** tableau;


    strcpy(resultat, chemin);
    strcat(resultat, "sorted.txt");


    fichier = fopen(chemin, "r");
```

```
    while (fgets(c, TAILLE_MAX_LIGNE, fichier) != NULL)
    {
        nombre_lignes++;
    }

    fseek(fichier, 0, SEEK_SET);

    tableau = (char**)malloc(nombre_lignes * sizeof(char*));

    while (fgets(ligne_lue, TAILLE_MAX_LIGNE, fichier) != NULL)
    {
        ligne_lue[strlen(ligne_lue)-1] = '\0';
        asprintf(&tableau[i], "%s", ligne_lue);
        i++;
    }

    qsort(tableau, nombre_lignes, sizeof(char *), cmp);

    output_fichier = fopen(resultat, "w");

    for (i = 0; i < nombre_lignes; i++)
    {
        printf("processing line... %s\n", tableau[i]);
        fprintf(output_fichier, "%s\n", tableau[i]);
    }

    printf("tri terminé! :-)\n");
    printf("le fichier %s contient %ld lignes! c'est cool\n", chemin, nombre_lignes);

    free(tableau);
    free(resultat);
    fclose(fichier);
    fclose(output_fichier);
}
```

/*

Fonction qui crée un dossier "partition" et y place un partitionnement du fichier dictionnaire donné en argument selon les classes grammaticales.

Fonction qui les détermine: partOfSpeech(), incluse dans searching.h.

verbe, adjectif, pronom, nom commun, nom propre, adverbe, conjonction de coordinations, préposition

v adj pro nc np adv coo prep

conjonction de subordination, determinant,

csu(afin que) cld(lui) det

```
*/
void partitionFile(char filename[])
{
    char ligne_lue[TAILLE_MAX_LIGNE];
    char* classeg = (char*)malloc(TAILLE_LIGNE_MAX * sizeof(char));
    FILE* dico = NULL;
    FILE* v = NULL;
    FILE* adj = NULL;
    FILE* pro = NULL;
    FILE* nc = NULL;
    FILE* np = NULL;
    FILE* adv = NULL;
    FILE* coo = NULL;
    FILE* prep = NULL;
    FILE* csu = NULL;
    FILE* cld = NULL;
    FILE* det = NULL;
    FILE* reste = NULL;

    dico = fopen(filename, "r");

    mkdir("partition", 0777);
    chdir("partition");

    v = fopen("v", "a");
    adj = fopen("adj", "a");
    pro = fopen("pro", "a");
    nc = fopen("nc", "a");
    np = fopen("np", "a");
    adv = fopen("adv", "a");
    coo = fopen("coo", "a");
    prep = fopen("prep", "a");
    csu = fopen("csu", "a");
    cld = fopen("cld", "a");
    det = fopen("det", "a");
    reste = fopen("reste", "a");

    while (fgets(ligne_lue, TAILLE_MAX_LIGNE, dico) != NULL)
    {
        printf("processing line: %s", ligne_lue);

        classeg = partOfSpeech(ligne_lue, classeg);

        if (strcmp(classeg, "v") == 0) {fprintf(v, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "adj") == 0) {fprintf(adj, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "pro") == 0) {fprintf(pro, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "nc") == 0)      {fprintf(nc, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "np") == 0)      {fprintf(np, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "adv") == 0) {fprintf(adv, "%s", ligne_lue); continue;}
        if (strcmp(classeg, "coo") == 0) {fprintf(coo, "%s", ligne_lue); continue;}
```

```
            if (strcmp(classeg, "prep") == 0) {fprintf(prep, "%s", ligne_lue); continue;}
            if (strcmp(classeg, "csu") == 0) {fprintf(csu, "%s", ligne_lue); continue;}
            if (strcmp(classeg, "cld") == 0) {fprintf(cld, "%s", ligne_lue); continue;}
            if (strcmp(classeg, "det") == 0) {fprintf(det, "%s", ligne_lue); continue;}

            fprintf(reste, "%s", ligne_lue);
    }
    fclose(dico);
    fclose(v);
    fclose(adj);
    fclose(pro);
    fclose(nc);
    fclose(np);
    fclose(adv);
    fclose(coo);
    fclose(prep);
    fclose(csu);
    fclose(cld);
    fclose(det);
    free(classeg);
    chdir("..");
}
```

```
int main()
{
    int choix;
    char* chemin;

    while (1)
    {
            choix = 0;
            chemin = (char*)calloc(TAILLE_MAX_LIGNE, sizeof(char));

            printf("\nBienvenue dans l'utilitaire de traitement de dictionnaires. Ce programme vous permet de:\n\n1. Parcourir votre fichier en remplaçant les caractères accentués (faisant partie de l'alphabet français) et les majuscules  par le caractère correpondant;\n");
            printf("2. Réorganiser votre fichier en classant ses lignes dans l'ordre croissant pour la fonction strcmp();\n3. Partitionner le dictionnaire en plusieurs fichiers correspondant chacun à une classe grammaticale;\n4. Quitter.\n\nEntrez le chiffre correspondant à votre choix: ");
            scanf("%d", &choix);
            printf("\n");

            if ((choix != 1) && (choix != 2) && (choix != 3) && (choix != 4))
            {
                    printf("Erreur: sélection invalide, veuillez recommencer.\n");
```

```
		free(chemin);
		return 1;
	}

	if (choix == 1)
	{
		printf("Le programme créera une copie de votre fichier dans le repertoire courant avec un suffixe \"std.txt\"; le fichier initial restera donc intact.\nPressez 1 pour commencer ou 0 pour quitter maintenant: ");
		scanf("%d", &choix);

		if ((choix != 1) && (choix != 0))
		{
			printf("Erreur: sélection invalide, veuillez recommencer.\n");
			free(chemin);
			return 1;
		}

		if (choix == 0)
		{
			printf("Au revoir....\n");
			free(chemin);
			return 0;
		}
		else
		{
			printf("Entrez le chemin de votre fichier: ");
			scanf("%s", chemin);
			standardizeFile(chemin);
			continue;
		}
	}

	if (choix == 2)
	{
		printf("ATTENTION! Cette procédure est effectuée sur le fichier d'origine; créez une copie de sauvegarde se besoin. Le processus peut prendre beaucoup de temps si les données sont volumineuses, mais peut être arrêté et repris ultérieurement.\nPressez 1 pour commencer ou 0 pour quitter maintenant :");
		scanf("%d", &choix);

		if ((choix != 1) && (choix != 0))
		{
			printf("Erreur: sélection invalide, veuillez recommencer.\n");
			free(chemin);
			return 1;
		}

		if (choix == 0)
		{
			printf("Au revoir....\n");
```

```
                free(chemin);
                return 0;
            }
            else
            {
                printf("Entrez le chemin de votre fichier: ");
                scanf("%s", chemin);
                sortFile(chemin);
                continue;
            }
        }

        if (choix == 3)
        {
            printf("Un dossier nommé \"partition\" sera créé dans le repertoire courant, dans lequel tous les fichiers seront placés. Le fichier d'origine restera intact.\nPressez 1 pour commencer ou 0 pour quitter maintenant: ");
            scanf("%d", &choix);

            if ((choix != 1) && (choix != 0))
            {
                printf("Erreur: sélection invalide, veuillez recommencer.\n");
                free(chemin);
                return 1;
            }

            if (choix == 0)
            {
                printf("Au revoir....\n");
                free(chemin);
                return 0;
            }
            else
            {
                printf("Entrez le chemin de votre fichier: ");
                scanf("%s", chemin);
                partitionFile(chemin);
                continue;
            }
        }

        if (choix == 4)
        {
            printf("Au revoir....\n");
            free(chemin);
            return 0;
        }
        free(chemin);
    }
}
```

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>



/*
ENCODAGE NECESSAIRE DE L'INPUT POUR LA FONCTION STANDARDIZE: ISO 8859-15
*/



/*
Extrait une partie du mot, comprise entre les indices i et j. On suppose:
i < j;
j <= (strlen(mot) - 1);
*/

char* extractPart(char mot[], int i, int j, char partie[])
{
    partie = strndup(mot+i, j+1);

    return partie;
}




/*
Fonction concaténant en un mot deux mots prix en argument.
*/
char* concateneMots(char mot1[], char mot2[], char concatenation[])
{
    sprintf(concatenation, "%s%s", mot1, mot2);

    return concatenation;
}
```

```
/*
Supprime majuscules et caractères accentués d'une chaine en les remplaçant par les caractères correspondants.
*/

char* standardizeMot(const char mot[], char result[])
{
    int i = 0;

    int length = strlen(mot);

/*  printf("mot = %s, longueur = %d\n", mot, length);*/
    for (i = 0; i <= length; i++)
    {
        result[i] = mot[i];
        if ((mot[i] == 'à') || (mot[i] == 'â')) {result[i] = 'a'; continue;}
        if ((mot[i] == 'é') || (mot[i] == 'è') || (mot[i] == 'ê') || (mot[i] == 'ë')){result[i] = 'e'; continue;}
        if ((mot[i] == 'ï') || (mot[i] == 'î')) {result[i] = 'i'; continue;}
        if (mot[i] == 'ô'){result[i] = 'o'; continue;}
        if (mot[i] == 'ç'){result[i] = 'c'; continue;}
        if ((mot[i] == 'ù') || (mot[i] == 'û')) {result[i] = 'u'; continue;}
        if (mot[i] == 'A') {result[i] = 'a'; continue;}
        if (mot[i] == 'B') {result[i] = 'b'; continue;}
        if (mot[i] == 'C') {result[i] = 'c'; continue;}
        if (mot[i] == 'D') {result[i] = 'd'; continue;}
        if (mot[i] == 'E') {result[i] = 'e'; continue;}
        if (mot[i] == 'F') {result[i] = 'f'; continue;}
        if (mot[i] == 'G') {result[i] = 'g'; continue;}
        if (mot[i] == 'H') {result[i] = 'h'; continue;}
        if (mot[i] == 'I') {result[i] = 'i'; continue;}
        if (mot[i] == 'J') {result[i] = 'j'; continue;}
        if (mot[i] == 'K') {result[i] = 'k'; continue;}
        if (mot[i] == 'L') {result[i] = 'l'; continue;}
        if (mot[i] == 'M') {result[i] = 'm'; continue;}
        if (mot[i] == 'N') {result[i] = 'n'; continue;}
        if (mot[i] == 'O') {result[i] = 'o'; continue;}
        if (mot[i] == 'P') {result[i] = 'p'; continue;}
        if (mot[i] == 'Q') {result[i] = 'q'; continue;}
        if (mot[i] == 'R') {result[i] = 'r'; continue;}
        if (mot[i] == 'S') {result[i] = 's'; continue;}
        if (mot[i] == 'T') {result[i] = 't'; continue;}
        if (mot[i] == 'U') {result[i] = 'u'; continue;}
        if (mot[i] == 'V') {result[i] = 'v'; continue;}
        if (mot[i] == 'W') {result[i] = 'w'; continue;}
        if (mot[i] == 'X') {result[i] = 'x'; continue;}
        if (mot[i] == 'Y') {result[i] = 'y'; continue;}
        if (mot[i] == 'Z') {result[i] = 'z'; continue;}
    }
/*  printf("mot obtenu: %s, de longueur: %d\n", result, strlen(result));*/
    return result;
```

```
}
```

```
a           auxavoir    [@favoir,@active,@p3s]
ainsi       csu   []
apres 120   prep  [pred='apres_____1<obj:sn|sa|sinf>',@pcasapres]
aucun       det   [det=+,define=-,@ms]
aucune      det   [det=+,define=-,@fs]
aussi       csu   []
avant 120   prep  [pred='avant_____1<obj:sn|sa>']
avec  120   prep  [pred='avec_____1<obj:sn|sa>',@pcasavec]
avt   120   prep  [pred='avant_____1<obj:sn|sa>']
because     csu   []
bicause     csu   [wh=+]
bref        csu   []
cad         coo   [pred='c'est-a-dire_____1<arg1,arg2>',cat = coo]
car         coo   [pred='car_____1<arg1,arg2>',cat = coo]
ce          det   [det=+,demonstrative=+,@ms]
ces         det   [det=+,demonstrative=+,@p]
cet         det   [det=+,demonstrative=+,@ms]
cette       det   [det=+,demonstrative=+,@fs]
cf    120   prep  [pred='cf_____1<obj:sn|sa>']
chaque      det   [det=+,define=-,@s]
chez  120   prep  [pred='chez_____1<obj:sn|sa>']
ci-devant   120   prep  [pred='ci-devant_____1<obj:sn|sa>']
comme       coo   [pred='comme_____1<arg1,arg2>',cat = coo]
comme       csu   []
comme       120   prep  [pred='comme_____1<obj:sn|sa>',@pcascomme]
confer      120   prep  [pred='cf_____1<obj:sn|sa>']
contre      120   prep  [pred='contre_____1<obj:sn|sa|sinf>',@pcascontre]
dans  120   prep  [pred='dans_____1<obj:sn|sa>',@pcasdans]
de          det   [det=+,define=-,@p]
de    120   prep  [pred='de_____1<obj:sn|sa|scompl|sinf>',@pcasde]
depuis      120   prep  [pred='depuis_____1<obj:sn|sa>']
derriere    120   prep  [pred='derriere_____1<obj:sn|sa>']
des         det   [det=+,define=-,@p]
des   120   prep  [pred='des_____1<obj:sn|sa>']
dessous     120   prep  [pred='dessous_____1<obj:sn|sa>']
dessus      120   prep  [pred='dessus_____1<obj:sn|sa>']
devant      120   prep  [pred='devant_____1<obj:sn|sa>',@pcasdevant]
devers      120   prep  [pred='devers_____1<obj:sn|sa>']
dixit 120   prep  [pred='dixit_____1<obj:sn|sa>']
donc        coo   [pred='donc_____1<arg1,arg2>',cat = coo]
donc        csu   []
du          det   [det=+,define=-,@ms]
durant      120   prep  [pred='durant_____1<obj:sn|sa>']
en    120   prep  [pred='en_____1<obj:sn|sa>',@pcasen]
endeans     120   prep  [pred='endeans_____1<obj:sn|sa>']
entre 120   prep  [pred='entre_____1<obj:sn|sa|sinf>']
envers      120   prep  [pred='envers_____1<obj:sn|sa>']
et          coo   [pred='et_____1<arg1,arg2>',cat = coo]
excepte     120   prep  [pred='excepte_____1<obj:sn|sa|sinf>']
hormis      120   prep  [pred='hormis_____1<obj:sn|sa|sinf>']
```

```
hors 120 prep [pred='hors_____1<obj:sn|sa|sinf>']
jusque 120 prep [pred='jusque_____1<obj:sn|sa>']
la det [det=+,define=+,@fs]
ladite det [det=+,define=+,@fs]
le det [det=+,define=+,@ms]
ledit det [det=+,define=+,@ms]
les det [det=+,define=+,@p]
les 120 prep [pred='les_____1<obj:sn|sa>']
lesdites det [det=+,define=+,@fp]
lesdits det [det=+,define=+,@mp]
leur cld [pred='pro',case=dat,@3p]
leur det [det=+,@poss,@s_p3p]
leurs det [det=+,@poss,@p_p3p]
lorsque csu [wh=+]
lui cld [pred='pro',case=dat,@3s]
ma det [det=+,@poss,@fs_p1s]
mais coo [pred='mais_____1<arg1,arg2>',cat = coo]
malgre 120 prep [pred='malgre_____1<obj:sn|sa>']
me cld [pred='pro',case=dat,@1s]
mes det [det=+,@poss,@p_p1s]
moi cld [pred='pro',case=dat,@1s]
mon det [det=+,@poss,@s_p1s]
moyennant 120 prep [pred='moyennant_____1<obj:sn|sa|sinf>']
ni coo [pred='ni_____1<arg1,arg2>',cat = coo]
nonobstant 120 prep [pred='nonobstant_____1<obj:sn|sa|sinf>']
nos det [det=+,@poss,@p_p1p]
notre det [det=+,@poss,@s_p1p]
nous cld [pred='pro',case=dat,@1p]
or coo [pred='or_____1<arg1,arg2>',cat = coo]
ou coo [pred='ou_____1<arg1,arg2>',cat = coo]
outre 120 prep [pred='outre_____1<obj:sn|sa|sinf>']
par 120 prep [pred='par_____1<obj:sn|sa|sinf>',@pcaspar]
par-dela 120 prep [pred='par-dela_____1<obj:sn|sa|sinf>']
par-dessous 120 prep [pred='par-dessous_____1<obj:sn|sa>']
par-dessus 120 prep [pred='par-dessus_____1<obj:sn|sa|sinf>']
parmi 120 prep [pred='parmi_____1<obj:sn|sa>']
partant csu []
paske csu [wh=+]
pcq csu [wh=+]
pdt 120 prep [pred='pendant_____1<obj:sn|sa>']
pendant 120 prep [pred='pendant_____1<obj:sn|sa>']
plus coo [pred='plus_____1<arg1,arg2>',cat = coo]
plusieurs det [det=+,define=-,@p]
pour 120 prep [pred='pour_____1<obj:sn|sa|sinf>',@pcaspour]
pr 120 prep [pred='pour_____1<obj:sn|sa|sinf>',@pcaspour]
puis coo [pred='puis_____1<arg1,arg2>',cat = coo]
puis csu []
puisque csu [wh=+]
qd csu [wh=+]
qq det [det=+,define=-]
qqes det [det=+,define=-,@p]
```

```
qqs         det   [det=+,define=-,@p]
quand             csu   [wh=+]
que         csu   [wh=+,que=+]
quel        det   [det=+,qu=+,@ms]
quelle            det   [det=+,qu=+,@fs]
quelles           det   [det=+,qu=+,@fp]
quelque           det   [det=+,define=-,@s]
quelques          det   [det=+,define=-,@p]
quels       det   [det=+,qu=+,@mp]
quoique           csu   [wh=+]
sa          det   [det=+,@poss,@fs_p3s]
sans   120  prep  [pred='sans_____1<obj:sn|sa|sinf>']
sauf   120  prep  [pred='sauf_____1<obj:sn|sa>']
selon  120  prep  [pred='selon_____1<obj:sn|sa>']
ses         det   [det=+,@poss,@p_p3s]
si          csu   []
sinon       coo   [pred='sinon_____1<arg1,arg2>',cat = coo]
sinon       csu   []
soit        csu   []
son         det   [det=+,@poss,@s_p3s]
sous   120  prep  [pred='sous_____1<obj:sn|sa>']
suivant     120   prep  [pred='suivant_____1<obj:sn|sa>']
sur    120  prep  [pred='sur_____1<obj:sn|sa>',@pcassur]
ta          det   [det=+,@poss,@fs_p2s]
te          cld   [pred='pro',case=dat,@2s]
tel         det   [det=+,define=+,@ms]
telle       det   [det=+,define=+,@fs]
telles      det   [det=+,define=+,@fp]
tels        det   [det=+,define=+,@mp]
tes         det   [det=+,@poss,@p_p2s]
toi         cld   [pred='pro',case=dat,@2s]
ton         det   [det=+,@poss,@s_p2s]
tous        det   [det=+,define=-,@mp]
tout        det   [det=+,define=-,@ms]
toute       det   [det=+,define=-,@fs]
toutes            det   [det=+,define=-,@fp]
un          det   [det=+,define=-,@ms]
une         det   [det=+,define=-,@fs]
vers   120  prep  [pred='vers_____1<obj:sn|sa|sinf>',@pcasvers]
versus      120   prep  [pred='vs_____1<obj:sn|sa|sinf>']
via    120  prep  [pred='via_____1<obj:sn|sa>']
vis-a-vis   120   prep  [pred='vis-a-vis_____1<obj:sn|sa>']
voire       coo   [pred='voire_____1<arg1,arg2>',cat = coo]
vos         det   [det=+,@poss,@p_p2p]
votre       det   [det=+,@poss,@s_p2p]
vous        cld   [pred='pro',case=dat,@2p]
vs          cld   [pred='pro',case=dat,@2p]
vs     120  prep  [pred='vs_____1<obj:sn|sa|sinf>']
```

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>



#define TAILLE_LIGNE_MAX 200




int nombreLignes(char chemin[])
{
    int nombre_lignes = 0;
    char c[TAILLE_LIGNE_MAX];

    FILE* fichier = NULL;
    fichier = fopen(chemin, "r");

    while (fgets(c, TAILLE_LIGNE_MAX, fichier) != NULL)
    {
        nombre_lignes++;
    }

    fclose(fichier);
    return nombre_lignes;
}







/*
accoutrée v [pred='accoutrer_____1<obj:(sn),objde:(de-sn),suj:(par-
sn)>',cat=v,@passive,@être,@Kfs]
*/

char* partOfSpeech(char ligne_dico[], char classeg[])
{
    char mot[TAILLE_LIGNE_MAX];
    int* nombre = malloc(sizeof(int));
    sscanf(ligne_dico, "%s %d %s", mot, nombre, classeg);
    if (*nombre == 0)
    {
        sscanf(ligne_dico, "%s %s", mot, classeg);
```

```
    }

    free(nombre);
    return classeg;
}

/*
verbe, adjectif, pronom, nom commun, (nom propre), adverbe, *conjonction de coordinations, *préposition
v adj pro nc np adv coo prep
conjonction de subordination, determinant,
csu(afin que) cld(lui) det

*/



/*
Fonction qui extrait le mot de la ligne du dictionnaire correspondante.
*/
char* getMot(char ligne_dico[], char mot_recup[])
{
    sscanf(ligne_dico, "%s", mot_recup);
    return mot_recup;
}




char* rechercheDichotomique(char mot[], FILE* dictionnaire, char ligne_renvoyee[])
{
    int stop = 0;
    int position_current = 0;
    int n = 2;
    int signe = 1;
    int etape = 0;
    int modificateur = 0;
    int k = 0;
    int l = 0;
    int leng = 0;
    int taille_dico;
    char aux[TAILLE_LIGNE_MAX];
    char mot_lu[TAILLE_LIGNE_MAX];
    char c[1];

    fseek(dictionnaire, 0, SEEK_END);
    taille_dico = ftell(dictionnaire);
```

```
	fseek(dictionnaire, 0, SEEK_SET);

	while (stop != 1)
	{
		if (position_current%2 == 0)
		{
			modificateur = taille_dico/n;
		}
		else
		{
			modificateur = ((int)taille_dico/(n)) + 1;
		}

		if (modificateur < k)
		{
			modificateur = k;
		}

		position_current = position_current + signe*(modificateur);

		if (position_current < 0)
		{
			position_current = position_current + modificateur - l;
		}


		etape = etape + 1;
		n = n * 2;

		fseek(dictionnaire, position_current, SEEK_SET);
		l = 0;

		while((c[0] = fgetc(dictionnaire)) != '\n')
			{
			fseek(dictionnaire, -2, SEEK_CUR);
			l = l + 1;

		}
		position_current = position_current - l;


		fscanf(dictionnaire, "%s", mot_lu);
		leng = strlen(mot_lu);
   /*
		printf("etape %d, position dans le fichier: %d, pas de la recherche: %d\n", etape,
position_current, modificateur);
		printf("mot lu: %s\n", mot_lu);
		printf("comparaison entre %s et '%s': %d\n\n", mot_lu, mot, strcmp(mot_lu,
mot));
   */
		if (etape > 25)
```

```
            {
                fseek(dictionnaire, 0, SEEK_SET);
                return NULL;
            }
            if (strcmp(mot_lu, mot) == 1)
            {
                signe = -1;
                k = 24;
                while ((c[0] = fgetc(dictionnaire)) != '\n')
                {
                    k = k + 1;
                }

                fseek(dictionnaire, -k, SEEK_CUR);

            }
            if (strcmp(mot_lu, mot) == -1)
            {
                signe = 1;
                k = 24;
                while ((c[0] = fgetc(dictionnaire)) != '\n')
                {
                    k = k + 1;
                }

                fseek(dictionnaire, -k, SEEK_CUR);

            }
            if (strcmp(mot_lu, mot) == 0)
            {
   /*           printf("%s = %s, recherche terminee en %d etapes!\n", mot_lu, mot,
etape);*/
                stop = 1;
            }
        }

        fseek(dictionnaire, -leng, SEEK_CUR);
        fgets(aux, TAILLE_LIGNE_MAX, dictionnaire);
        strcpy(ligne_renvoyee, aux);
   /*
        printf("taille du fichier: %d octets\n", taille_dico);
   */
        fseek(dictionnaire, 0, SEEK_SET);
        return ligne_renvoyee;
   }
```

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>


#define DICO "dictionnaire.txt"
#define LINK "link_words.txt"
#define OUTPUT "resultats.txt"
#define TAILLE_LIGNE_MAX 200




int nombreLignes(const char chemin[])
{
    int nombre_lignes = 0;
    char c[200];

    FILE* fichier = NULL;
    fichier = fopen(chemin, "r");

    while (fgets(c, TAILLE_LIGNE_MAX, fichier) != NULL)
    {
        nombre_lignes++;
    }

    fclose(fichier);
    return nombre_lignes;
}






/*
accoutrée v [pred='accoutrer_____1<obj:(sn),objde:(de-sn),suj:(par-
sn)>',cat=v,@passive,@être,@Kfs]
*/

char* partOfSpeech(char ligne_dico[], char classeg[])
{
    char mot[TAILLE_LIGNE_MAX];
    int* nombre = malloc(sizeof(int));
    sscanf(ligne_dico, "%s %d %s", mot, nombre, classeg);
    if (*nombre == 0)
```

```
    {
        sscanf(ligne_dico, "%s %s", mot, classeg);

    }

    free(nombre);
    return classeg;
}

/*
verbe, adjectif, pronom, nom commun, (nom propre), adverbe, *conjonction de coordinations, *préposition
v  adj  pro  nc  np  adv  coo  prep
conjonction de subordination,  determinant,
csu(afin que)  cld(lui)  det

*/




char* rechercheDichotomique(char mot[], FILE* dictionnaire, char ligne_renvoyee[])
{
    int stop = 0;
    int position_current = 0;
    int n = 2;
    int signe = 1;
    int etape = 0;
    int modificateur = 0;
    int k = 0;
    int l = 0;
    int leng = 0;
    int taille_dico;
    int pos_prec;
    char aux[TAILLE_LIGNE_MAX];
    char mot_lu[TAILLE_LIGNE_MAX];
    char c[1];

/*  printf("recherche de\"%s\" lancée\n", mot);*/

    fseek(dictionnaire, 0, SEEK_END);
    taille_dico = ftell(dictionnaire);
    fseek(dictionnaire, 0, SEEK_SET);

    while (stop != 1)
    {
        if (position_current%2 == 0)
        {
```

```
                modificateur = taille_dico/n;
        }
        else
        {
                modificateur = ((int)taille_dico/(n)) + 1;
        }

        if (modificateur < k)
        {
                modificateur = k;
        }

        pos_prec = position_current;
        position_current = position_current + signe*(modificateur);

        if (position_current < 0)
        {
                position_current = (int)(pos_prec/2);
        }

        if (position_current >= taille_dico)
        {
                position_current = pos_prec + (int)((taille_dico - pos_prec)/2);
        }

        etape = etape + 1;
        n = n * 2;

        fseek(dictionnaire, position_current, SEEK_SET);
        l = 0;

        while((c[0] = fgetc(dictionnaire)) != '\n')
                {
                fseek(dictionnaire, -2, SEEK_CUR);
                l = l + 1;

        }
        position_current = position_current - l;


        fscanf(dictionnaire, "%s", mot_lu);
        leng = strlen(mot_lu);
   /*
        printf("etape %d, position dans le fichier: %d, pas de la recherche: %d\n", etape,
position_current, modificateur);
        printf("mot lu: %s\n", mot_lu);
        printf("comparaison entre %s et '%s': %d\n\n", mot_lu, mot, strcmp(mot_lu,
mot));
   */
        if (etape > 30)
        {
```

```
                fseek(dictionnaire, 0, SEEK_SET);
                return NULL;
            }
            if (strcmp(mot_lu, mot) == 1)
            {
                signe = -1;
                k = 24;
                while ((c[0] = fgetc(dictionnaire)) != '\n')
                {
                    k = k + 1;
                }

                fseek(dictionnaire, -k, SEEK_CUR);
                continue;

            }
            if (strcmp(mot_lu, mot) == -1)
            {
                signe = 1;
                k = 24;
                while ((c[0] = fgetc(dictionnaire)) != '\n')
                {
                    k = k + 1;
                }

                fseek(dictionnaire, -k, SEEK_CUR);
                continue;

            }
            if (strcmp(mot_lu, mot) == 0)
            {
    /*              printf("%s = %s, recherche terminee en %d etapes!\n", mot_lu, mot,
etape);*/
                stop = 1;
            }
        }

        fseek(dictionnaire, -leng, SEEK_CUR);
        fgets(aux, TAILLE_LIGNE_MAX, dictionnaire);
        strcpy(ligne_renvoyee, aux);
    /*
        printf("taille du fichier: %d octets\n", taille_dico);
    */
        fseek(dictionnaire, 0, SEEK_SET);
        return ligne_renvoyee;
    }
```

```
void chercheAmbiguite(char mot[], char exception1[], char exception2[], FILE* dictionnaire, FILE* dictionnaire_link, FILE* fichier_resultat, char ligne_renvoyee[])
{
    int i = 0;
    int k = 0;
    int length = strlen(mot);
    char* part1;
    char* part2;
    char* part3;
    char* link_word;

    for (i = 0; i < length; i++)
    {
        part1 = strndup(mot, i+1);
        part2 = strndup(mot+i+1, length-1);

        if ((strcmp(part1, exception1) != 0) && (strcmp(part2, exception2) != 0))
        {
/*
            printf("%s %s\n", part1, part2);
*/
            if (rechercheDichotomique(part1, dictionnaire, ligne_renvoyee) != NULL)
            {
                if (rechercheDichotomique(part2, dictionnaire, ligne_renvoyee) != NULL)
                {
                    printf("%s %s <<< concaténation initiale\n%s %s <<< partitionnement alternatif; URL équivoque valable!\n", exception1, exception2, part1, part2);
                }

                if (i < length-1)
                {
                    for (k = 1; k < (length-1-i); k++)
                    {
                        link_word = strndup(mot+i+1, k);
                        part3 = strndup(mot+i+k+1, length-1);
```

```
                                    if ((rechercheDichotomique(part3, dictionnaire, ligne_renvoyee) != NULL) && (rechercheDichotomique(link_word, dictionnaire_link, ligne_renvoyee) != NULL))
                                    {
                                        printf("%s %s  <<< concaténation initiale\n%s %s %s  <<< partitionnement alternatif; URL équivoque valable!\n", exception1, exception2, part1, link_word, part3);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

```
int main(int argc, char ** argv)
{
    int i = 0;
    int nombre_lignes;
    char* result;
    char* result2;
    char ligne_lue[TAILLE_LIGNE_MAX];
    char* mot_recup = (char*)malloc(TAILLE_LIGNE_MAX * sizeof(char));
    char* concatenation = (char*)malloc(TAILLE_LIGNE_MAX * sizeof(char));
    FILE* fichier = NULL;
    FILE* dictionnaire_link = NULL;
    FILE* fichier_resultat = NULL;

/*Déclaration des variables relatives à rechercheDichotomique */
    FILE* FICHIER_DICO;
    char* ligne_renvoyee = (char*)malloc(TAILLE_LIGNE_MAX * sizeof(char));

/*Déclaration des variables relatives à partOfSpeech */
    char* classeg = (char*)malloc(TAILLE_LIGNE_MAX * sizeof(char));
```

```
	if (argc != 2)
	{
		printf("Un seul mot en argument à la fois, merci.\n");
		free(mot_recup);
		free(ligne_renvoyee);
		free(concatenation);
		free(classeg);
		return 0;
	}

	FICHIER_DICO = fopen(DICO, "r");

	result = rechercheDichotomique(argv[1], FICHIER_DICO, ligne_renvoyee);

	if (result != NULL)
	{
		result2 = partOfSpeech(result, classeg);
		printf("%s", result);
		printf("Classe grammaticale = %s\n", result2);
	}
	else
	{
		printf("Le mot propose ne fait pas partie du dictionnaire :'-)\n");
	}

	nombre_lignes = nombreLignes(DICO);

	fichier = fopen(DICO, "r");
	dictionnaire_link = fopen(LINK, "r");
	fichier_resultat = fopen(OUTPUT, "w");
  /*  fprintf(fichier_resultat, "%s", "Voici les resultats de la recherche:\n");*/

	for (i = 0; i < nombre_lignes; i++)
	{
		fgets(ligne_lue, TAILLE_LIGNE_MAX, fichier);
		sscanf(ligne_lue, "%s", mot_recup);
		if (i%1000 == 0)
		{
			printf("%dème mot lu: %s\n", i, mot_recup);
		}
		sprintf(concatenation, "%s%s", argv[1], mot_recup);
		chercheAmbiguite(concatenation, argv[1], mot_recup, FICHIER_DICO,
dictionnaire_link, fichier_resultat, ligne_renvoyee);
		sprintf(concatenation, "%s%s", mot_recup, argv[1]);
		chercheAmbiguite(concatenation, mot_recup, argv[1], FICHIER_DICO,
dictionnaire_link, fichier_resultat, ligne_renvoyee);
	}

	printf("Recherche terminée! :'-)\n");

	fclose(FICHIER_DICO);
```

```
    fclose(fichier);
    free(ligne_renvoyee);
    free(concatenation);
    free(classeg);
    return 0;
}
```

MATCH () La fonction qui va chercher les termes équivoques dans les deux mots fournis.
Il faut bien lancer la fonction avec match(source, motDico) puis match(motDico, source)

ENTREE : string source le mot entré par l'utilisateur
string motDico le mot proposé par le dictionnaire

RETOUR : rien, le fichier texte qui contient les résultat à été incrémenté des réponses valides trouvées.

Debut algo :

```
chaine chaine1 ="", chaine2=""; chaineTest=""
integer position1, position2

// Dans le sens source puis md

position1<- taille(source)-1
position2 <- 0


//tant qu'on, est pas arrivé à la fiun d'un des deux mots
Tant que (taille(chaine1) < taille(source))&&(taille(chaine2) < taille(motDico)))
    Faire :
    position1 --
    position2 ++

    chaine1 <- source[position1->fin] (substr())
    chaine2 <- motDico[0 -> position2]

    //si c'est exactement le même mot
    Si (position1 = taille(source) && position2= taille(motDico))
        Faire:
        sortir de la boucle
    Fin Si

    Sinon si : (chaine1 = chaine2) (strcmp())
        Faire :
        chaineTest = motDico[position2 ->fin]
        si chaineTest estPresent() dans le dictionnaire
            Faire:
            ouvrir fichierResultat
            ecrire à la suite :
                source[0->position1] - source[position1+1->fin]  - motDico[position2->fin]
            fermer fichierResultat
        Fin Si
    Fin si
Fin Tant Que
```