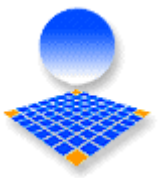
Ministère de l'Éducation Nationale

Université de Montpellier II

Rapport de Projet Informatique ULIN 607
de la Licence informatique 3ème année effectué
de janvier 2010 à mai 2010

Chemins Mnémotechniques

Berthelot Victor
Bianciotto Romain
Bontemps Jean-Charles
Jourdan Bérenger



Remerciements

Nous remercions en premier lieu notre tuteur Philippe Gambette pour son aide précieuse tout au long du semestre.

Nous remercions également JeuxDeMots et en particulier Mathieu Lafourcade pour la mise à disposition de la base de données.

Nous remercions le département informatique qui nous permet un accès facile aux salles et matériels informatiques.

Nous remercions également nos différents professeurs d'algorithmiques qui nous ont permis de travailler efficacement.

Table des Matières

1. Introduction

1.1 Généralités

Nous sommes un groupe de quatre étudiants en troisième année de licence d'informatique à l'université de Montpellier 2. Pour notre dernier semestre d'étude on nous a demandé de choisir un projet informatique à élaborer.

C'est ainsi que vers la fin du mois de janvier nous avons commencé à réfléchir sur le projet des chemins mnémotechniques. Nous avons eu environ 3 mois pour le mener à terme et faire une soutenance orale le 19 mai.

Durant ces trois mois, nous avons été sous la surveillance de notre tuteur Philippe Gambette qui a su répondre à nos questions quand il le fallait et qui nous a guidé sur certains points assez flou.

1.2 Le sujet

Il s'agit de créer un outil pour aider à trouver des “chemins mnémotechniques” pour retenir du vocabulaire dans une langue étrangère. Il s'agit donc de retenir l'association entre deux mots, par exemple “travel” et “voyage”, en construisant un chemin de mots entre les deux.

Ce chemin de mots commence par associer au mot anglais un mot français dont la graphie est proche. Par exemple pour “travel” : un “travelling” ou une “traversée”. Puis le chemin procède par associations d'idées en français : travel → travelling → cinéma → siège → train → voyage.

Nous verrons plus tard que pour des raisons d'efficacité le programme n'utilise pas ce raisonnement au niveau du code. Cela ne change rien au principe d'utilisation du programme.

Comme source de données on utilisera la base de données de JeuxDeMots. C'est un fichier texte rempli de mot français et de relations entre eux en fonction des données fourni par les internautes jouant à JeuxDeMots.
La base de données est à récupérer sur : http://www.lirmm.fr/~lafourcade/JDM-LEXICALNET-FR/

Par exemple, on peut avoir un aperçu des mots associés à “voyage” de la base de JeuxDeMots sur la page http://www.lirmm.fr/jeuxdemots/rezo.php

Une fois le programme fini cela doit être accessible avec l'aide d'une interface web.

1.3 Cahier des charges

Pour coder l'arbre de recherche, les différentes fonctions de recherches et les fonctions de comparaisons, nous utiliserons le C.

C'est un langage multiplateforme, qui permet d'avoir un code compréhensible par d'autres personnes tout en gardant une bonne efficacité.

Le projet sera hébergé sous : http://sourceforge.net/projects/cheminsmnmotech/

Cela nous a permis d'utiliser le SVN suivant :
https://cheminsmnmotech.svn.sourceforge.net/svnroot/cheminsmnmotech

L'interface web sera codée globalement en HTML, puis XHTML, Javascript pour les sections n'ayant besoin d'aucun traitement spécifiques.

Nous avons utilisé le framework JQUERY qui permet de créer un code plus simple, plus clair entre les pages de l'interface. Aussi un rafraichissement via AJAX à été plus facile à implémenter.

La page du programme générant le graphique sera elle codée avec du PHP qui nous sert à traiter le formulaire et appeler le programme avec les bons paramètres vérifiés.

Nous allons maintenant exposer les différentes parties prévues dans le programme.

a) L'implémentation de la base de données

Comme il est précisé dans le sujet, on nous a fourni un fichier texte qui nous servira de base de données pour trouver des relations lexicales.
Ce fichier texte est divisé en deux grandes parties :

- La première où sont représentés les nœuds du graphe (identifiant, mot, type, poids).
- La deuxième partie représente les relations entre les différents nœuds (par identifiant, type, poids).

La première partie du programme consiste donc à rendre traitable ce fichier texte. Pour commencer on charge en mémoire le fichier de base de données (pour un traitement plus rapide). Cela prend déjà ~60Mo.

La base de données de JeuxDeMots est légèrement bruitée c'est à dire qu'on y trouve des erreurs, des noms de mots voulant dire quelque chose de non documenté actuellement, il a fallu pour cela filtrer au préalable les mots.
La structure utilisée en mémoire est une structure d'arbre, cela permet une recherche en O(1) théorique, l'inconvénient de cette méthode est une place mémoire qui peut être considérée comme importante : 50Mo pour l'arbre. (plus 50Mo pour l'arbre inverse : nom > eid)
Le traitement se fait en moins de 4 secondes sur un 1.6Ghz (Windows).
Pour l'instant on pourrait donc utiliser ce logiciel sans forcément avoir besoin d'un daemon (logiciel lancé en temps que Service), ou d'une utilisation client / serveur.

Pour l'instant la mémoire maximum nécessitée pour un tel traitement est de 170Mo. On peut prévoir qu'au final en estimant que la base de données de JeuxDeMots grossisse jusqu'à 300-400Mo, cela nécessiterait peut-être jusqu'à 1.2Go de mémoire ce qui est correct encore pour des ordinateurs récents. (pour des serveurs ou même des ordinateurs personnels).

Une fois l'implémentation de la base de données terminée on commencera à s'intéresser aux liaisons reliant deux mots dans le chemin mnémotechnique. On commencera par traiter la liaison principale, celle qui relie au mot étranger (mot qui n'apparait pas dans la base) à un mot proche graphiquement apparaissant dans la base de données.

b) La liaison utilisant la proximité graphique entre deux mots :

Dans un premier temps, on utilisera une mesure classique de distance entre mots : la distance de Levenshtein.

En tout il y aura 6 distances de codées :

Levenshtein , Jaro , Jaro-Winkler , SoundEx , DoubleMetaphone , SoundEx + Jaro-Winkler , DoubleMetaphone + Jaro-Winkler.

Ces autres distances seront développées plus loin dans le rapport.

La distance de Levenshtein calcule le nombre d'opérations minimales pour passer d'un mot à un autre.
Ces opérations sont: la suppression, l'ajout et le remplacement d'un caractère.

Exemple :
Travel => traversée.

Ces deux mots sont proches graphiquement. En effectuant un remplacement et trois ajouts de caractères sur le premier mot, on obtient le second mot.

Donc, pour cette relation, la distance vaut 4. Afin de trouver rapidement les mots proches graphiquement, on va utiliser deux filtres successifs:

Le premier filtre permettra de ne garder que des mots ayant le même préfixe/suffixe que le mot de départ.

Afin d'effectuer ce filtre, la base de données sera récrite sous forme d'arbre trie. L'arbre trie est un arbre dont chaque branche correspond à chacun des caractères possibles d'un mot. (a,b,c,d...). Ainsi, si le mot commence par un 'd', on prend la branche correspondant à la lettre 'd'.

On a alors accès à tous les mots commençant seulement par la lettre 'd'. A partir de là, si on ne veut garder que les mots commençant par "da", on prendra la branche correspondant au caractère 'a'. Et ainsi de suite...

Pour ce premier filtre, on ne gardera qu'une cinquantaine de mots.
Le second filtre utilisera la distance de Levenshtein afin de ne garder que cinq mots sur les cinquante.

Plus la distance entre 2 mots est courte, plus ces mots sont proches graphiquement. Les cinq mots choisis seront donc ceux qui auront la distance la plus courte avec le mot de départ.

Pour avoir une harmonie avec les autres distances codées nous avons du ramener cette distance à un pourcentage.

Ce pourcentage de similarité est calculé à partir de la formule: 1-(distance/(somme des longueurs des mots/2).

Ainsi, si la distance vaut 0, le pourcentage de similarité est de 100% sachant que la taille des 2 mots varient, faire une moyenne de ce pourcentage est plus approprié pour le calcul car le résultat pourrait varier si on inversait les 2 mots.

Dans le cas contraire, si le pourcentage de similarité obtenu est négatif, alors il est mis à 0%.

c) Les chemins lexicaux :

Cette étape permet de relier le mot d'origine à traduire au mot proche graphiquement du mot à traduire que l'on a obtenu grâce aux différentes distances.

Dans cette étape, on parcourt les relations entres les mots de la base de données afin de trouver le chemin le plus court et efficace possible, on effectuera un parcours en largeur sur une certaines hauteur à partir des mots retournés par la proximité graphique en l'orientant vers un mot cible.

Pour cela on utilisera les informations de la base de données pour chaque mot ; les types et les poids des relations d'un mot d'une part ainsi que le nombre de voisins et la proximité graphique entre deux mots voisins pour définir quelles relations parcourir en priorité.

Ce parcours ne dépassera pas une hauteur donnée, d'abord pour ne pas obtenir des chemins inutilement long, mais surtout pour limiter les temps de calcul des différents chemins.

Enfin, on pourra chercher à améliorer les chemins obtenus, par exemple, en regroupant les mots trop proches graphiquement afin d'obtenir des chemins plus courts, et donc plus intéressants.

d) Création des chemins :

Après analyse du problème il nous a semblé plus logique de partir du mot connu (dans la base de données) pour trouver ensuite à partir de ses relations un lien avec le mot anglais, plutôt que de devoir rechercher dans la base toutes les correspondances avec le mot anglais.

Concrètement cela signifie que l'on part du mot connu et qui existe dans la base de données. On construit d'abord l'arbre complet jusqu'à la hauteur maximale demandée. Après nous lancerons une fonction de filtrage qui permettra d'affiner l'affichage avec le nombre de mot total souhaité.

Par exemple si nous avons demandé 15 chemins, le programme regardera dans tous les chemins créés quel sont les 15 meilleurs chemins en fonction du pourcentage de similarité obtenu.

En effet cette technique ne nécessite pas de calculer la distance par rapport à tout les autres mots. C'est à dire que si on partait du mot anglais, on devrait comparer au début ce mot à tous les mot qui lui ressemble graphiquement et ensuite chercher le mot français voulu. Cette technique serait beaucoup moins rapide.

Lors de la création du graphique nous sommes obligés de nous limiter à une certaine hauteur, d'une part pour obtenir un résultat facile à mémoriser, mais aussi pour éviter d'avoir des calculs trop importants.

e) L'interface :

Il faudra donc créer une interface web permettant à un utilisateur d'utiliser cet outil.
Dans un formulaire l'utilisateur saisit le mot de départ et le mot-cible. Il doit ensuite sélectionner une distance graphique qui permet de relier le mot étranger à un mot présent dans la base de données. Il devra ensuite choisir une distance maximale pour les chemins renvoyés ainsi que le nombre total de chemins voulu.

Nous pensons dans un premier temps afficher le graphe avec un générateur automatique de graphe (par exemple dot).
Cette technique réduirait le rendu final car un générateur comme dot créer immédiatement l'image et il n'y pas de possibilité de prise de contrôle avant la fin de la génération.
Nous essaierons donc de proposer une autre manière de générer le graphe de telle façon à ce que l'interaction soit possible.
On utilisera probablement la technologie HTML MAP pour définir des zones cliquables et pour l'arbre nous utiliserons soit la librairie GD de PHP ou la technologie SVG

Une fois la zone cliquée, le programme sera de nouveau exécuté avec de nouveaux paramètres.

f) Améliorations envisagées :

Voici différents objectifs qui permettront d'améliorer l'application suivant le temps restant.

La principale amélioration consiste à améliorer le filtrage des mots lors de la recherche d'un chemin. En effet, le fichier source attribue à chaque mot un nombre important de relations qui sont parfois peu intéressantes et quelques fois inutilisables.

Ces fonctions de filtrage nous permettraient d'éviter de renvoyer une relation peu efficace dans un chemin.

L'autre objectif secondaire que nous souhaiterions améliorer concerne l'interface web et l'interaction avec le graphe.
Nous avons développé l'explication de cette amélioration dans le paragraphe concernant l'interface.
Nous pourrions aussi envisager de travailler avec une base de données plus importante, car celle fournie est en cours de développement et non complète.

g) Licence :

L'outil créé sera un logiciel libre sous licence GPL. Il pourra donc être repris par quiconque souhaitant apporter des améliorations à l'outil.

Par conséquent, le code doit être suffisamment détaillé et suffisamment propre afin de faciliter leur tâche.

h) Répartition du travail :

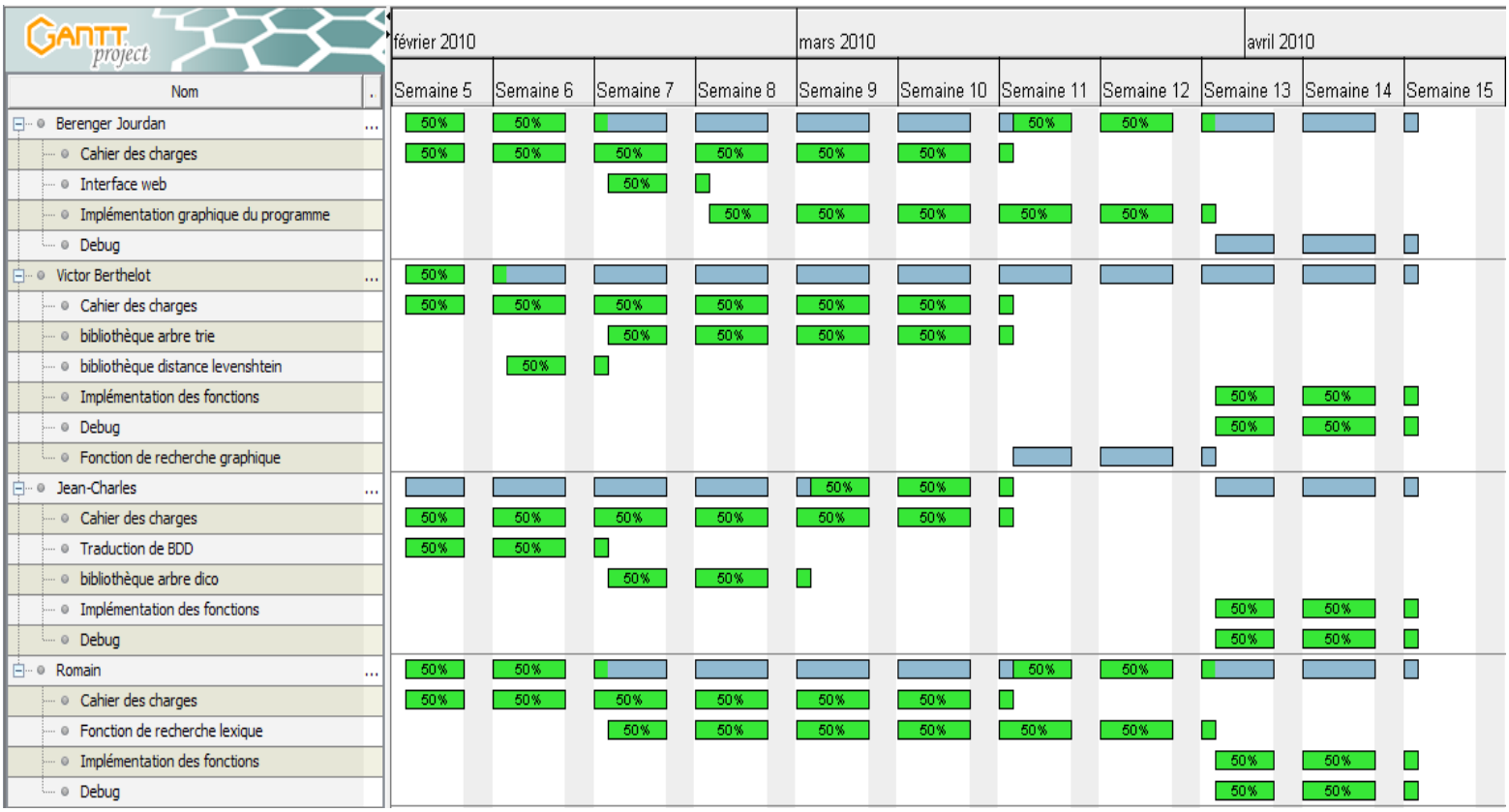
- Implémentation de la base de données: Jean-Charles Bontemps

- Implémentation de la proximité graphique : Victor Berthelot et Romain Bianciotto.

- Implémentations des chemins lexicaux : Victor Berthelot et Romain Bianciotto.

- Interface web et génération du graphique : Bérenger Jourdan

i) Diagrammes de GANTT :



2. Organisation du projet

2.1 Organisation du travail

Au début du semestre nous avons mis en place des réunions hebdomadaires tous les mardi matins, la plupart en présence de notre tuteur pour bien mettre en place les bases du projet. Ces réunions se sont toutes passées dans une des salles du bâtiment 6 préalablement réservée par notre tuteur.

Durant les trois premières semaines du semestre, nous nous sommes donc vu tous les mardi matins pour analyser le sujet du projet. On a discuté des structures de données à utiliser pour les points cruciaux du programme, de la manière d'implémenter la base de données pour permettre une recherche efficace (quel type d'arbre utiliser, contenu de chaque nœuds …) et surtout un des points les plus importants : l'implémentation de la création de chemin a partir de la base de données qui fut un débat (étape durant laquelle nous avons été en désaccord). Nous avons aussi commencé à analyser la manière de relier le programme à l'interface web. Nous verrons que nous avions eu plein de bonnes idées que malheureusement nous n'avons pas réussi à implémenter pour différentes raisons.

Une fois cette première analyse terminée nous avons commencé à discuter de la répartition des tâches. Nous avons établi état des lieux des capacités des membres du groupe ainsi que les préférences de chacun pour choisir quelle partie du projet coder. Contrairement à certain groupes nous n'avons pas choisi de chef de projet. Cela ne nous a pas posé de problèmes d'organisation comme notre groupe est restreint a quatre membres. La plupart du temps quand il fallait faire un choix important nous tombions souvent d'accord.

Après ces séances d'analyse du projet, nous avons établis trois groupes de travail :

- un premier groupe d'une personne chargée d'implémenter la base de données sous forme d'arbre. Ce travail comprend la création de chemins utilisant les relations lexicographiques présentes dans la base de données.

- un deuxième de deux personnes chargées d'implémenter les différentes distances graphiques. Ce groupe est en relation avec le premier afin d'assurer une cohérence entre les implémentations des distances graphiques crées et les chemins fournis.

- un dernier groupe d'une personne devant s'occuper de l'interface web et des différentes manières de représenter graphiquement les chemins retournés.

Durant le reste du semestre nous nous sommes rencontrés toutes les trois ou quatre semaines environ pour mettre en commun nos avancées. Les rencontres n'ont pas toujours été évidentes à organiser pour causes d'emplois du temps très différents.

2.2 Choix des outils de développement

Le programme en lui même est complètement codé en C sans ajout de bibliothèque particulière. Il est compilable avec la dernière version stable de GCC. Chacun utilise son éditeur préféré, cela n'interfère pas dans la réalisation du projet.

L'interface web utilise plusieurs langages :

-Le traitement des paramètres est contrôlé avec du PHP.
-La partie graphique utilise quelques fonctionnalités du framework JQUERY et de l'AJAX.

Pour la démonstration, nous allons héberger temporairement la version finale sur un serveur http.
Nous fournirons également une version en ligne de commandes qui permet de faire des tests un peu plus complet, car l'interface web supporte difficilement une génération comprenant un nombre trop importants de chemins.

Durant notre projet nous avons utilisé le gestionnaire de version SVN hébergé sur sourceforge.net. Côté client nous utilisons le logiciel tortoiseSVN sous Windows, ou Subversion sous Unix.
Pendant la durée du projet nous avons également communiqué entre nous par mail, et par des logiciels de messageries instantanées de type MSN.

3. Analyse du projet

Rappel rapide du sujet :

Le but du projet consiste à fournir un moyen mnémotechnique permettant de retenir la traduction d'un mot français en une langue étrangère. Ce moyen mnémotechnique est une association de mots représentant un chemin mnémotechnique entre le mot français et sa traduction. Pour nous aider nous avons à disposition une base de données qui contient une liste de mots et de relations entre ces mots.

Comme nous l'avons expliqué dans le cahier des charges, les données sont stockées sous la forme d'un gigantesque arbre dans un fichier texte.

Le fichier texte est divisé en deux grandes parties :

-La première où sont représentés les nœuds du graphe (identifiant, mot, type, poids).

Ces nœuds sont représentés avec quatre paramètres de cette manière :

eid=720:n="absolu":t=1:w=72

–eid représente l'identificateur unique du mot.

–n représente la chaîne de caractères

–t le type de mot. Reportez-vous à la section node stats, présente au début du fichier txt de la base de données pour connaître les différents type de mots.

–W le poids du mot (correspond à la fréquence d'apparition dans JeuxDeMots. Cela peut correspondre à sa fréquence d'utilisation dans la langue Française). Plus le nombre d'apparition du mot est important plus sont poids augmente.

La base de données comporte environ 208000 mots dont 36 000 noms propres.
(Malheureusement beaucoup sont inutiles : ::>13:123769>29:131333>14 ou serviette>63999" … Ce genre de mots a peu d'intérêt à être utilisé dans des moyens mnémotechniques.)

– La deuxième partie représente les relations entre les différents nœuds (par identifiant, type, poids). Ces relations sont représentées de la manière suivante :

rid=27804:n1=134635:n2=146885:t=4:w=50

–rid représente l'identificateur unique de la relation

–n1 un mot d'un nœud.

–n2 un des voisins de n1.

–t le type de relation

–w le poids de la relation.

Note : la relation n'est pas toujours réciproque. n2 est un voisin de n1 par un certain type de relation mais l'inverse n'est pas toujours vrai.

La structure de base de données fournie dans un fichier texte n'est pas traitable rapidement.
Pour rechercher les relations d'un mot à partir de sa chaine de caractères, on devrait parcourir une fois le fichier, pour trouver l'eid, puis rechercher tout les eids le liant, pour enfin devoir retrouver les chaines de caractères correspondant à tout ces eid. Cela est bien trop long à traiter via un accès au disque dur.

La première tache consiste à rendre cette base plus rapide d'accès pour notre futur programme. La structure d'arbre nous a semblé appropriée pour ce problème.

Chaque nœud de l'arbre comprendra des informations fournis par le fichier texte :
- l'eid du mot (identificateur).
- son type
- son poids (fréquence d'apparitions dans JeuxDeMots)
- la liste de ses voisins.

On peut remarquer que contrairement à la base de données originale, nous intégrons directement la liste des voisins dans le nœud de ce nouvel arbre.

Une fois ce travail effectué, il sera donc possible d'implémenter une recherche dans l'arbre créé. Il nous restera donc à effectuer le lien entre un mot français apparaissant dans le chemin en cours de construction et le mot étranger n'apparaissant pas dans la base de données. Pour cela nous nous sommes documentés sur la notion de distance graphique.

Après quelques recherches nous avons décidé d'implémenter 3 distances graphiques : Levenshtein , Jaro , Jaro-Winkler .

A ces distances nous avons rajouté des distances phonétique : SoundEx , DoubleMetaphone.

.

Associer une distance phonétique au mot à rechercher permet d'avoir comme base une phonétique proche pour ensuite cibler encore plus précisément sur l'orthographe.

En cas d'un nombre trop importants de chemins, il faudra implémenter une fonction de filtrage permettant de garder les chemins les plus intéressants.

Nous pensons développer un système de client serveur avec l'interface web pour ne générer qu'une seule fois l'arbre et des données relatives aux chemins dont on pourra éditer les paramètres via l'interface.

Parallèlement un membre du groupe développera une interface graphique permettant l'exécution du programme final avec les paramètres adéquats.

Il nous semble que ce moyen mnémotechnique soit peu efficace. En effet selon certaines études (L'expérimentation et la démarche scientifique d'André TRICOT) cette méthode ne conviendrait pas à un utilisateur ayant un niveau avancée dans le langage voulu.

4. Traitement de la base de données et création des chemins

4.1 Traitement de la base de données

Tout d'abord les besoins sont :

-mémoriser un mot
-à ce mot est associé quelques variables (type, poids,eid)
-mémoriser des relations entre ces mots avec encore un (type, poids, rid)
-les relations sont unique et non réversibles (le lien a un seul sens)

Le fichier utilisé comme support est le fichier : "03152010-LEXICALNET-JEUXDEMOTS-FR-NOHTML.txt".

Pour que le calcul des distance soit plus correct il faut impérativement ne pas avoir d'HTML, sinon nous avons des problèmes avec les caractères spéciaux, accents.

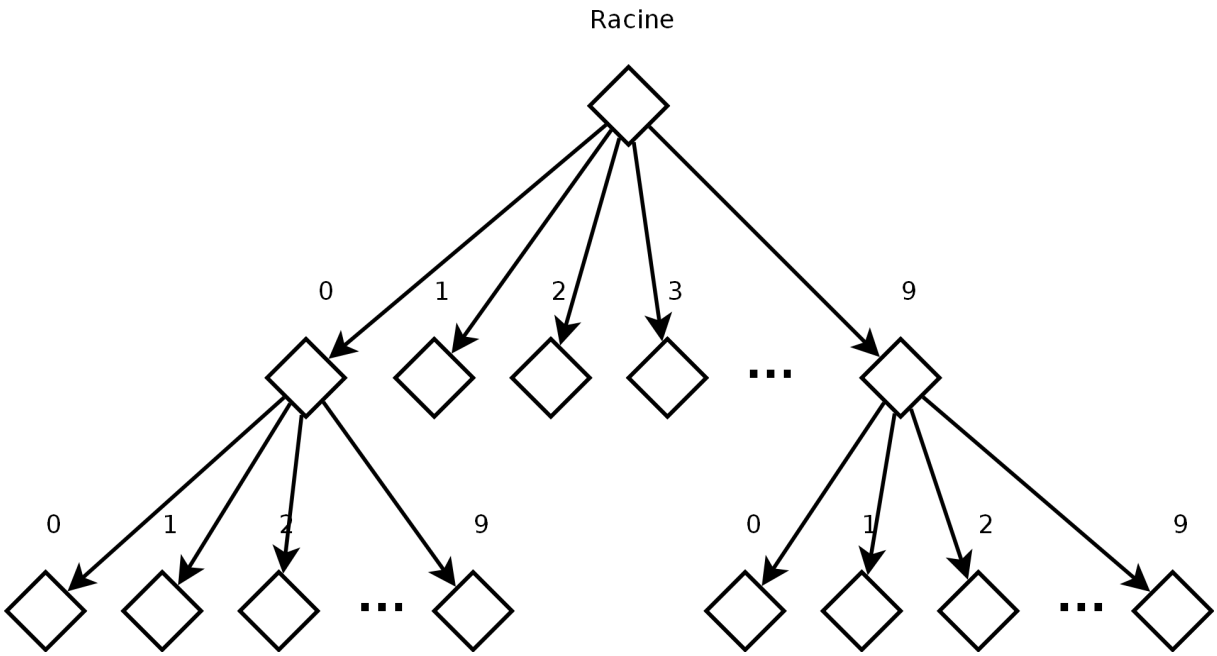
La base de données contient deux structures distinctes :
-les Nodes
-les Relations

Le Node est défini par un eid (identifiant), un type, un poids et un mot (chaîne de caractères).

Une relation est définie par un rid, un type, un poids, et deux eid.
La relation est à sens unique, c'est à dire qu'un mot X peut être en relation avec Y mais pas l'inverse.

Par soucis de rapidité, garder en mémoire la base entière parait une bonne idée.
Pour cela on s'est penché sur une structure d'arbre :

http://fr.wikipedia.org/wiki/Arbre_%28informatique%29



Ce n'est pas un arbre binaire, mais un arbre de base 10, ce qui permet de rechercher un eid car écrit en base 10 dans le fichier fournit.

On peut donc une fois l'arbre crée retrouver un Node très rapidement à partir de son eid.

On aura besoin pour le calcul des chemins d'obtenir une liste des relations, on a donc choisis de les stocker dans une liste chainée. L'ajout se fait très rapidement O(1), le parcours se fait normalement O(n).

On attribut donc à chaque Neux de l'arbre un "Node"
Le Node contient un type, poids, mot, et une "Liste de relations".
Une Liste de relations est une simple liste chaînée, contient un pointeur vers le suivant, un mot, un type, un poids.

Pour pouvoir retrouver un mot dans l'arbre à partir de son nom (chaîne de caractères), il faut créer encore un arbre pour pouvoir le trouver tout aussi rapidement.

Par soucis de compromis temps / mémoire, calculer le hash du mot parait être une bonne solution, cela permet de garder exactement la même structure d'arbre en faisant pointer les Nodes de ce nouvel arbre vers les Nodes de l'autre arbre pour profiter de ce qui a déjà été mémorisé.

La fonction de hash utilisée renvoie à partir d'une chaine de caractères un int32 ce qui permet d'avoir 4 G-hash différents, les chances d'avoir des collisions sont donc minimes pour la base utilisée. On peut mettre plus de 7000 fois ce volume de données avant d'atteindre la moitié des possibilités. Aussi la hauteur de l'arbre ne sera pas plus grande que 10 car 4294967296 (2^32) ne contient que 10 caractères. On gagne ainsi en place.

Lors de l'analyse du fichier fourni, on ajoute les relations aux listes de relations des Nodes, une option a été rajoutée permettant de choisir le sens d'ajout de ces relations.
Par exemple une relation : **mot1 -> mot2** peut alors être rajoutée à la base dans l'autre sens :

mot1 <- mot2 mais encore la dernière option permet d'ajouter les deux sens : **mot1 <---> mot2**

L'avantage est qu'une fois chargé, la rapidité permet de calculer les chemins dans un temps acceptable pour un affichage graphique via une interface web par exemple.

L'inconvénient de cette méthode est de devoir stocker un grand nombre d'informations en mémoire RAM.

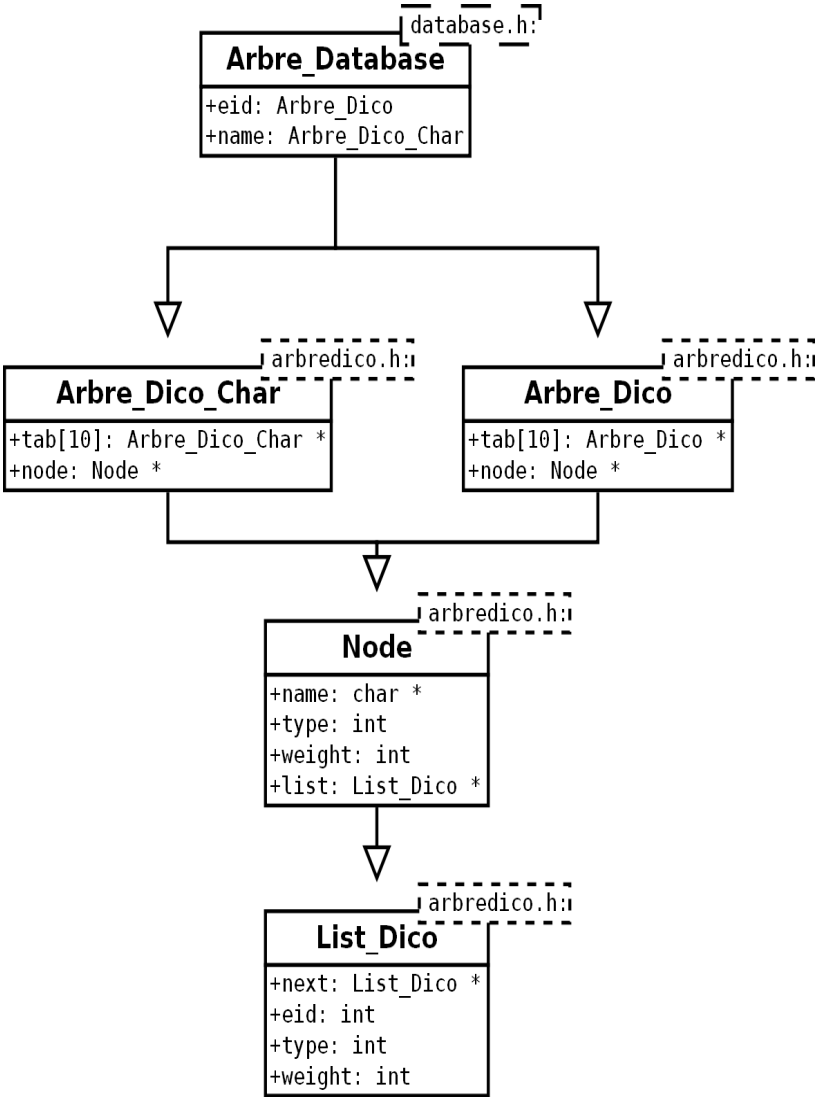
Devoir garder le fichier fourni, et les deux arbres font qu'on a au total besoin de trois fois la taille du fichier fourni pour pouvoir exécuter le programme.

Ici ~60Mo de BDD, l'utilisation de Ram monte à 170Mo au total.

On a en détail 170Mo ~ répartis ainsi :
-15 Mo Ressources nécessaires à l'exécution du programme, environnement C etc...
-60 Mo BDD
-95 Mo Arbre eid -> Nodes + Arbre nom -> Nodes

Ces valeurs sont abstraites uniquement pour donner une ordre d'idée.

Schéma simplifié :



4.2 Création des chemins.

Les chemins sont stockés dans une structure en forme d'arbre n-aire.

Les nœuds contiennent :
-Node
-Liste de chemins (liste chaînée)
-les informations relatives à la distance par rapport au mot recherché qui sont calculées au moment de la création de l'arbre des chemins.

Un graphique plus normal ne devrait pas être un simple arbre, il devrait y avoir des relations entre les branches. Nous ne les avons pas crées car elles ne nous sont pas utiles pour ce que nous recherchons, aussi cela complexifierais beaucoup les algorithmes qui sont déjà lents.

La hauteur fait que les calculs suivent une fonction proche de l'exponentielle, dépendant bien sur du nombre de relations de chaque mot.

Aussi la création de l'arbre des chemins se fait par étapes, chaque hauteur est calculée après avoir calculée la hauteur précédente. Lorsque l'on rencontre un mot déjà présent dans le graphique on ne le rajoute donc pas car un chemin menant à lui plus court à déjà été trouvé, cela permet aussi de limiter les calculs. Par la même occasion, l'arbre ainsi créé ne peut pas être plus grand que la base de données elle-même.

Une fois l'arbre des chemins créé, nous nous sommes basé sur un unique critère de sélection pour l'instant : obtenir au minimum x mots dans notre graphique, et ceux-ci étant bien sur le plus similaire possible au mot recherché.

Pour une meilleure visualisation nous avons implémenté une fonction permettant de spécifier à DOT la couleur en fonction de la distance.

La commande est : "-c:x" avec x appartenant à : {-1, 0,1}
"-1" n'affiche pas les couleurs.
"0" affiche un dégradé entre rouge pour une distance proche de 0 et vert pour une distance proche de 1.
"1" affiche un dégradé entre bleu proche de 0 et vert proche de 1.

Les similarités sont calculées via les distances (Levenshtein, Jaro, Jaro-Winkler et les conjugaisons d'algorithmes phonétiques comme SoundEx et Double-Métaphone).

Pour trouver la distance limite à laquelle il ne faut plus prendre en compte les mots, on parcourt plusieurs fois l'arbre des chemins afin de compter le nombre de mots ayant une distance élevée, puis en trouvant la distance juste inférieure cela nous permet d'augmenter peu à peu le nombre de mots jusqu'à trouver le nombre désiré (ou tout afficher).

Lorsque l'on calcul la distance phonétique de SoundEx et de Double-Métaphone, cela renvoie simplement une chaine de caractères correspondant à un hash phonétique, c'est un peu comme si on avais appliqué une fonction de compression sur le mot. Cela est codé sur seulement 4 caractères via SoundEx ce qui crée beaucoup de collisions, on a donc plusieurs mots bien différents qui ont au final exactement le même résultat.

C'est pourquoi nous avons décidé d'améliorer la notion de similarité en rajoutant à la fin de ce hash le mot correspondant. En y appliquant l'algorithme Jaro-Winkler nous avons donc le préfixe correspondant au hash de l'algorithme phonétique, suivit du mot calculé en distance de Jaro simple.

Cela permet donc dans un premier temps de regrouper les mots par distance phonétique, suivi d'un classement par une distance de Jaro simple.

On pourrais trouver une multitudes de fonctions nous donnant des similarités.

Par exemple "-dist:snj" utilise SoundEx + Jaro-Winkler de la sorte :
"travel" est transformé en SoundEx("travel")+"---"+"travel"
soit : "T614----travel"
les "----" entre travel et le hash sont la pour tromper la fonction Jaro-Winkler d'une certaine manière. Jaro-Winkler calcule les changements dans les chaines en fonction de la distance.

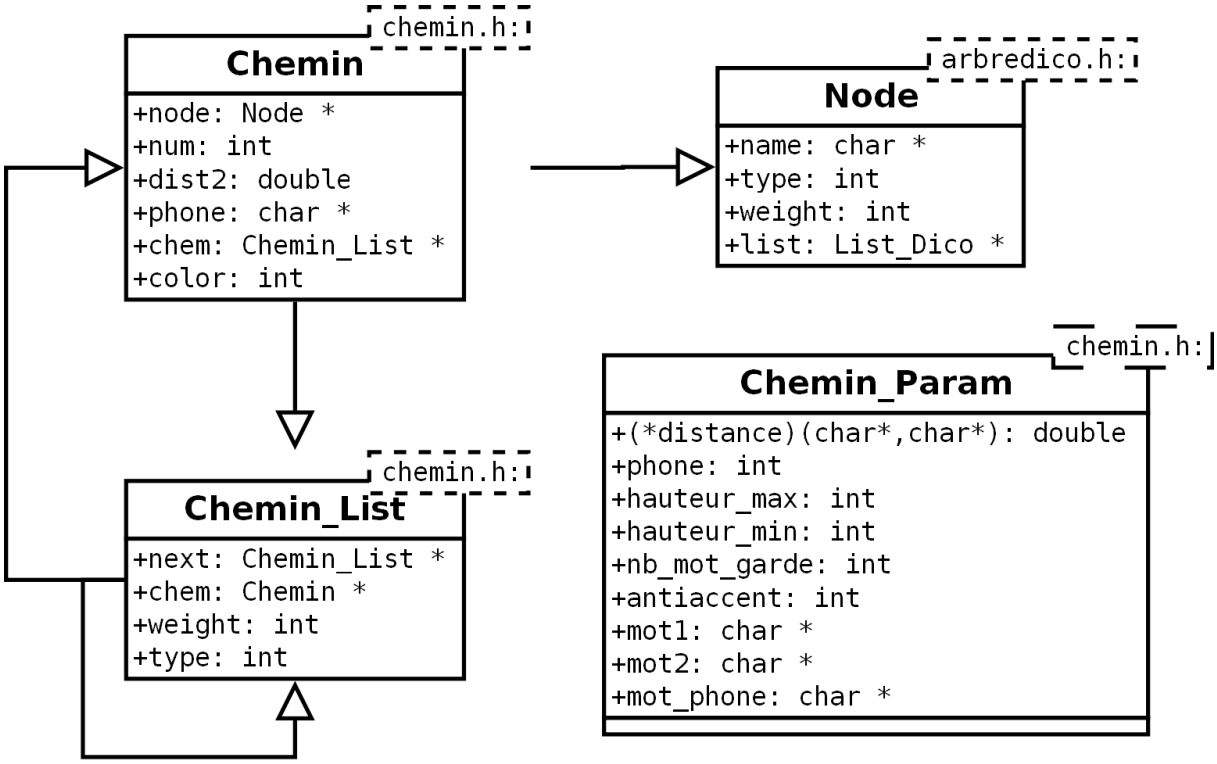
En éloignant le mot du hash, cela permet de ne pas assimiler de lettres du hash avec le mot en question.

Ces méthodes ont été conçues pour pouvoir calculer ultérieurement une hauteur supplémentaire à l'arbre des chemins, par exemple on pourrais afficher un premier résultat de hauteur 2 ou 3 assez rapidement, puis au fur et a mesure compléter le graphique.

On pourrait même via AJAX, afficher "graphique en cours de création", et montrer l'évolution, spécifier de progresser la recherche en fonction de la hauteur, ou modifier tout autre paramètre en temps réel.
Le problème est que le logiciel n'accepte pas pour l'instant une relation serveur / client.
Cela est une des évolutions prévues.

On a réussit à générer l'arbre des chemins en parallèle sur plusieurs cœurs, pour accroître la rapidité.
Pourquoi pas aussi générer de cette manière l'arbre des Nodes / Relations, mais cela est moins évident à implémenter.
Cela permettrait d'accélérer les calculs sur les ordinateurs récents ainsi que sur les serveurs.

Schéma simplifié des structures des chemins :



5. Les différentes distances

5.1 Les distances graphiques

a) La distance de Levenshtein

La distance de Levenshtein est une distance graphique tout comme la distance de Jaro. Elle à été créée en 1965 par Vladimir Levenshtein.

Cette distance est une généralisation de la distance de Hamming qui elle, ne peut être effectuée que sur des mots de même longueur.

Son principe est de calculer le nombre minimal de suppressions, ajouts et substitutions de caractères pour passer d'un mot à un autre.

Voici un exemple pour illustrer cette formule.

s1 = travel
s2 = train

Pour transformer s1 en s2, on peut déjà voir à vue d'œil qu'il nous faudra 3 opérations afin d'arriver à s2. Voici un exemple possible en 3 étapes:

-travel
Substitution du v en i:
-traiel
Substitution du e en n:
-trainl
Suppression du l:
-train

Calculer la distance de Levenshtein permet de garantir l'optimalité du résultat. Ici, nous avons vu que la distance de Levenshtein entre travel et train était de 3. Maintenant, voyons voir si la formule retourne bien le même résultat.

Nous allons réaliser une matrice des couts des 2 mots.

Un coût correspond à une opération (ajout, suppression, remplacement).
Le but de cette matrice est de trouver le coût minimal pour aller de s1 à s2 en fonction des coûts précédemment calculés.

Dans un premier temps, on initialise la matrice des couts comme ceci:

	t	r	a	v	e	l
t	0	1	2	3	4	5
r	1	-	-	-	-	-
a	2	-	-	-	-	-
i	3	-	-	-	-	-
n	4	-	-	-	-	-

Voici le code d'initialisation de la matrice:

```
for(k=0;k<n;k++)
  d[k]=k;
for(k=0;k<m;k++)
  d[k*n]=k;
```

Ensuite, nous allons remplir les cases vides au fur et à mesure.
d[x,y]= min(d[x-1,y]+1, d[x,y-1]+1, d[x-1,y-1]+c)
c=0 si s1(x-1)=s2(y-1)
c=1 sinon

En fait, cela revient au principe de calculer la distance de Levenshtein pour tous les sous-mots préfixes de s1 et s2.
Voici le code:

```
for(i=1;i<n;i++)
  for(j=1;j<m;j++)
  {
    //calcul du coût de substitution
    if(s[i-1]==t[j-1])
      cost=0;
    else
      cost=1;
    //calcul du coût minimum
    d[j*n+i]=minimum(d[(j-1)*n+i]+1,d[j*n+i-1]+1,d[(j-1)*n+i-1]+cost);
  }
```

Au début, nous calculons le coût minimal de "tr" et "tr",
Pour ca nous avons besoin du coût minimal de "t" et "t", de "tr" et "t" et enfin de "t" et "tr". Nous avons déja ces coûts puisque ce sont des valeurs de départ de la matrice.
cout("tr","tr") = minimum(cout("t","tr")+1,cout("tr","t")+1,cout("t","t") +0)
cout("tr","tr") = minimum(1+1,1+1,0+0)
cout("tr","tr") = 0
On peut constater que ce résultat est cohérent, car "tr"="tr" donc la distance de Levenshtein vaut bien 0.

Ensuite, nous calculons le coût de "tra" et "tr", ensuite celui de "trav" et "tr", etc...
On arrive à la fin au calcul du coût minimal de "travel" et "train".

On obtient alors la matrice suivante:

	t	r	a	v	e	l
t	0	1	2	3	4	5
r	1	0	1	2	3	4
a	2	1	0	1	2	3
i	3	2	1	1	2	3
n	4	3	2	2	2	3

La distance de Levenshtein de "travel" et "train" correspond à la case d[n*m-1], autrement dit, la dernière case de la matrice, soit 3.

cout("travel","train") = minimum(cout("trave","train")+1,cout("travel","trai")+1,cout("trave","trai") +1)
cout("travel","train") = minimum(2+1,3+1,2+1)
cout("travel","train") = 3

L'inconvénient de la distance de Levenshtein, c'est qu'elle n'est pas utilisable en elle même, puisqu'elle ne retourne pas un pourcentage de similarité contrairement aux autres distances.

C'est pourquoi nous avons modifié celle ci afin qu'elle retourne un pourcentage de similarité en fonction de la longueur des mots.

Nous calculons donc la distance de Levenshtein comme vu ci dessus.

Ensuite nous calculons le pourcentage de similarité, qui est calculé à partir de la formule:

1-(d/(|s1|+|s2|/2).

d = distance de Levenshtein entre s1 et s2.
|s1| et |s2| = longueurs respectives des mots s1 et s2.

Voyons là en détails :

En fait, il s'agit de transformer la distance de Levenshtein en pourcentage de similarité (donc compris entre 0 et 1).

Pourquoi ne pas utiliser tout simplement la formule :

1-(d/|s1|) ou alors 1-(d/|s2|)

Le problème avec cette formule :

d/|s1| et d/|s2| peuvent avoir un résultat différent si |s1| et |s2| ne sont pas égaux (contrairement à Hamming).

Par conséquent, en faisant la moyenne des 2 longueurs, on obtient un résultat unique, même si l'on inverse s1 et s2:

Donc, avec cette méthode, Levenshtein(s1,s2) = Levenshtein(s2,s1).

Un exemple :

s1=oh
s2=rance
d=5
|s1|=2
|s2|=5

On applique les 2 formules :

d/|s1|=5/2 = 2.5
d/|s2|=5/5 = 1

Les 2 résultats sont nettement différents après inversion de s1 et s2

d/(|s1|+|s2|/2)=5/((2+5)/2)=5/(7/2) = 10/7
d/(|s2|+|s1|/2)=5/((5+2)/2)=5/(7/2) = 10/7

Les deux résultats sont identiques après inversion de s1 et s2.

Par conséquent, plutôt que d'utiliser une formule dont sa valeur varie en fonction de l'ordre de l'appel de ses arguments, nous avons préféré utiliser celle qui ne varie pas, afin de réduire sensiblement la marge d'erreurs.

Plus d/(|s1|+|s2|/2) est petit, plus le pourcentage de similarité est élevé.

Donc, en effectuant sa fonction inverse, le résultat de la fonction devient approximativement proportionnel au pourcentage de similarité, ce que l'on souhaitait au départ.

Note: si le résultat retourné par la formule est inférieur à 0%, on admettra que le pourcentage de similarité est suffisamment faible pour que celui-ci soit considéré comme nul (chose possible si la distance de Levenshtein est supérieure à la moyenne des longueurs de s1 et s2).

Cela permet de certifier aux fonctions qui utiliseront Levenshtein que le résultat retourné est forcément compris entre 0 et 1.

b) La distance de Jaro :

La distance de Jaro est une distance graphique, c'est à dire elle mesure la similarité entre deux chaînes de caractères. Elle a été créée en 1989 par Matthew A. Jaro. *(wikipedia.fr)*

Voici sa formule mathématique brute :

$$d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$$

m : est le nombre de caractères correspondants.
T : le nombre de transpositions
|s1| |s2|: longueur des deux mots à comparer

Voici un exemple pour illustrer ce calcul :

S1 = travel
S2 = train

On obtient rapidement la longueur des mots en c en faisant strlen (S1), strlen (S2).

Calcul de m :

Deux caractères identiques de s_1 et de s_2 sont considérés comme *correspondants* si leur éloignement (i.e. la différence entre leurs positions dans leurs chaînes respectives) ne dépasse pas :

-(max(|S1|,|S2|)/2)-1;

Concrètement pour calculer m il y a plusieurs étapes :

-Dans un premier temps on calcul l'écart max acceptable.
-Dans un second temps on créer une matrice de correspondance. On affecte 0 à l'intersection de deux caractères identiques.

Matrice de correspondance :

	t	r	a	v	e	l
t	0	1	1	1	1	1
r	1	0	1	1	1	1
a	1	1	0	1	1	1
i	1	1	1	1	1	1
n	1	1	1	1	1	1

Une fois que notre matrice est terminée il ne nous reste plus qu'a compter le nombre de 0.
A chaque 0 on regarde bien si la différence de position dans leur chaine respective ne dépasse pas l'écart max. Ici on a de la chance les caractères en commun (t,r,a) on une position identique. On en déduit donc que m vaut 3 ici.

Pour comparer voici le code en c du calcul de m :

Code c

```
for (i=0;i<l1;i++)
{
        for (j=max(i-ecartMax,0);j<=min(i+ecartMax,l2);j++)
        {
                if (t1[i]==t2[j])
                {
                        b1[i]=0; //Indique qu'on a bien trouvé ce caractère
                        b2[j]=0;
                        compteMatching++; //Incrémente le nombre de caractères correspondants (m)
                        break;
                }

        }

}
```

Il ne nous reste plus qu'à calculer t :

Pour calculer T nous utilisons la matrice de correspondance déjà créée car nous avons besoin des caractères communs.

Dans notre exemple nous avons donc vu qu'il y avait 3 caractères communs. (t,r,a).

Attention l'ordre est important. Ici, dans travel et train ces caractères apparaissent dans le même ordre mais ce n'est pas toujours le cas.

Par exemple avec les chaînes martha et marhta les caractères en commun sont m,a,r,t,h,a pour martha et m,a,r,h,t,a pour marhta. Cela aura une incidence sur la distance finale.

Dans notre programme on obtient cette liste de caractères en commun grâce à ce code :

code c

```
char *TrouverMatches(char * txt,int *bl)
{
//renvoie la chaine contenant les caractères communs
    int i,j;
    char *res=malloc(256*sizeof(char)); //un mot ne doit pas dépasser 256 caractères
    char ctmp='a';
    for (i=0;i<256;i++)
    {res[i]=0;}
    i=0,j=0;
    while (ctmp!=0)
    {
        ctmp=txt[i];
        if (bl[i]==0) //caractère commun.
        {
            res[j]=ctmp;
            j++;
        }
        i++;
    }
    return res;
}
```

Le premier paramètre correspond au mot dans lequel on doit trouver les caractères communs. Le second correspond à la matrice de correspondance.

Maintenant que nous avons obtenu la liste des caractères en commun il ne nous reste plus qu'à compter le nombre de transpositions.
Pour notre exemple avec travel et train il y a 0 transposition. (tra et tra sont identiques).

Par contre pour le cas avec martha et marhta il y a une transposition.(le h est échangé avec le t).

Voici le code qui compte le nombre de transposition :

code c :

```
compteTransposition=0;
    if (strcmp(t1Matche,t2Matche)!=0)
    {
        for (i=0;i<strlen(t1Matche);i++)
            if (t1Matche[i]!=t2Matche[i])
                compteTransposition++; //Calcul le nombre de transpositions
    }
    else
        compteTransposition=0;
```

T1matche et t2Match sont les chaînes de caractères en communs entre les deux mots initiaux.

Nous avons donc tous les éléments pour calculer la distance de Jaro entre deux mots.

La distance est de Jaro entre travel et train est donc de 0,7 et de 0,944 entre martha et marhta.

Note : Dans un souci d'efficacité la formule du programme C a été factorisée le plus possible pour limiter le nombre de division qui est une opération assez lourde en C.

c) Jaro-Winkler

La distance de Jaro-Winkler variante proposée en 1999 par William E. Winkler, découlant de la distance de Jaro. (wikipedia).

La méthode introduite par Winkler utilise un coefficient de préfixe p qui favorise les chaînes commençant par un préfixe de longueur L (avec L<4).

En considérant deux chaînes s1 et s2, leur distance de Jaro-Winkler dw est :

$$d_w = d_j + (\ell p(1 - d_j))$$

- d_j est la distance de Jaro entre s_1 et s_2
- ℓ est la longueur du préfixe commun (maximum 4 caractères)
- p est un coefficient qui permet de favoriser les chaînes avec un préfixe commun. Winkler propose pour valeur $p = 0.1$

Calcul de L :

Il suffit juste de compter la taille du préfixe en commun inférieur à 4.

L'implémentation en C donne :

code C

```
longueurPrefix=0;
    for (i=0;i<min(3,min(l1,l2))+1;i++) //longueur max : 4
    {
            if (t1[i]==t2[i])
                    longueurPrefix++;
            else
                    break;

    }
```

Maintenant comparons le résultat des distances de Jaro et Jaro-Winkler

Jaro(travel,train)=0,7
Jaro-Winkler(travel,train)=0,9

jaro(martha,marhta)=0,944
jaro-Winkler(martha,marhta)=0,9611

jaro(voyage,agenda)=0,444
jaro-Winkler(voyage,agenda)=0,444

On voit donc bien que la distance de Jaro-Winkler favorise sensiblement les chaînes de caractères avec des préfixes en commun. Ceci est intéressant car un préfixe en commun facilite quand même la mémorisation d'un chemin mnémotechnique.

On peut donc dire que la distance de Jaro-Winkler défini en partie la notion d'efficacité (une relation entre deux qui mot qui ont un préfixe commun est donc plus efficace) dont nous parlions dans le cahier des charges.

5.2 Les distances phonétiques.

Les distances phonétiques contrairement aux distances graphiques calcule les similarités non pas orthographique mais de sons produits dans la langue voulu. Ces deux algorithmes renvoient des hash de la phonétique des mots.

L'algorithme exact procède comme suit (wikipedia.fr) :

1. Supprimer les éventuels 'espace' initiaux
2. Mettre le mot en majuscule
3. Garder la première lettre
4. Conserver la première lettre de la chaîne
5. Supprimer toutes les occurrences des lettres : a, e, h, i, o, u, w, y (à moins que ce ne soit la première lettre du nom)
6. Attribuer une valeur numérique aux lettres restantes de la manière suivante :
    - Version pour l'anglais :
        - 1 = B, F, P, V
        - 2 = C, G, J, K, Q, S, X, Z
        - 3 = D, T
        - 4 = L
        - 5 = M, N
        - 6 = R
    - Version pour le français :
        - 1 = B, P
        - 2 = C, K, Q
        - 3 = D, T
        - 4 = L
        - 5 = M, N
        - 6 = R
        - 7 = G, J
        - 8 = X, Z, S
        - 9 = F, V
7. Si deux lettres (ou plus) avec le même nombre sont adjacentes dans le nom d'origine, ou s'il n'y a qu'un h ou un w entre elles, alors on ne retient que la première de ces lettres.
8. Renvoyer les quatre premiers octets complétés par des zéros.

En effectuant cet algorithme, on obtient avec "Robert" et "Rupert" la même chaîne : "R163", tandis que "Rubin" donne "R150".

La distance double Méthaphone est une amélioration de la distance Métaphone qui elle même est censé corriger les défauts de SoundEx.

Nous ne développerons pas plus ce paragraphe car comme c'est écris dans les en têtes de fichiers, nous ne sommes pas les auteurs de l'implémentation des deux distances phonétiques.

Dans notre cas nous utilisons le classement des consonnes à l'anglaise pour fabriquer les codes phonétiques.

6. Interface graphique

Pour la navigation du site nous avons choisi d'utiliser le framework JQUERY. Celui-ci nous permet de créer une architecture de site avec très peu de lignes de codes qui permettent d'associer chaque situées dans la div tabs à une nouvelle page du site.

De plus jquery est fourni avec quelques paramétrages en CSS, ce qui nous permet de nous concentrer sur des aspects plus importants de l'interface web.

a) Le formulaire

Nous utilisons un formulaire classique comme le propose le HTML pour permettre à l'utilisateur de rentrer tous les paramètres nécessaires à la génération d'un graphe. Une fois le formulaire complété et après validation de l'utilisateur, une fonction javascript vérifie les différents champs.

Une fois cette vérification terminée, on construit dans une variable l'url qu'on va passer en paramètre grâce à la méthode GET de la page responsable d'appeler le programme qui générera le fichier en format DOT.

Pendant cette génération on utilise la fonction ajax de jquery qui permet d'intégrer à la page déjà active le résultat du logiciel, et ce en ayant précédemment affiché le texte "Graphique en construction...
Attendez-svp".

b) L'appel du programme

La page php responsable de l'exécution du programme est donc appelée avec tous les paramètres adéquats (grâce à la méthode GET).

Dans un premier temps on supprime si besoin les fichiers graph.txt et graph.png pour éviter les confusions au cas ou le programme ne marcherais pas (cas de paramètres incorrect comme par exemple le mon français non présent dans la base de donnée).

Ensuite vient l'exécution du programme :

```
exec("main -fr:$var2 -en:$var1 -dist:$dist -fout:graph.txt -hmax:$hmax -hmin:1 -nb:$nb -c:0 -sens:$sens -c:$c -del -v > $log");
```

On exécute donc grâce à la fonction exec le programme comme si il était en mode console.

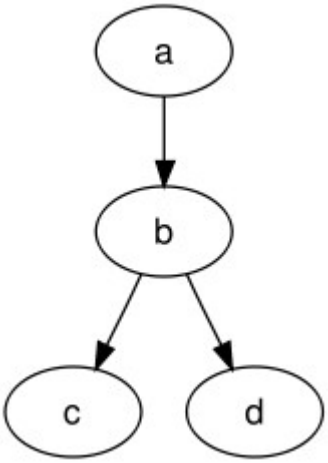
Remarque : la fin de l'appel > $log permet de stocker tous les paramètres initiaux dans un fichier texte ($log étant le chemin relatif d'un fichier texte). En prenant seulement le résultat de la commande exec, celle-ci nous retourne uniquement la dernière ligne du Stdout, on aurais pas le détail fourni par la commande -v.

Le programme à donc générer un fichier texte comprenant les différents chemins écris en format DOT (on dit également langage DOT).

c) Le format dot.

Le langage dot permet la création de graphe dans un fichier texte.

Voici un exemple simple :



Ce graphe est obtenu après le traitement d'un fichier texte contenant les lignes :

```
digraph mon_graphe
{
    a -> b;
    b -> c;
    b -> d;
}
```

Pour générer le graphe sous forme d'image nous avons choisi d'utiliser le programme dot qui fait parti de la suite d'outil graphviz (logiciel libre distribué suivant la licence Common Public licence http://fr.-wikipedia.org/wiki/Graphviz).

Voici un exemple d'appel du programme dot :

```
exec("dot -Gcharset=latin1 -Tpng graph.txt -o graph.png");
```

-Gcharset=latin1 est option nécessaire qui concerne les polices de caractères.

-Tpng permet de choisir le format du graphe de sorti (Dot supporte les formats images les plus populaires).

-graph.txt fichier d'entrée

-graph.png fichier de sortie.

d) Affichage du graphe.

Maintenant que l'image du graphe est créée, il ne reste plus qu'à l'afficher. Dans le cahier des charges, il est spécifié que l'interface devra être utilisée comme si elle était mise en ligne. Cela suppose que plusieurs utilisateurs peuvent l'utiliser en même temps. Cela pourrait générer quelques conflits lors de l'affichage d'une image.

Pour cela on utilise le fichier de sortie du programme comme jeton, on le supprime après avoir affiché son contenu, si jamais un utilisateur voudrais lancer le programme en même temps qu'un autre, le script php de création vérifierait alors que ce fichier existe déjà et afficherait un message comme quoi il faut réessayer plus tard.

7. Manuel d'utilisation

Nous fournissons deux parties dans notre rendu final. Une interface web servant d'interface graphique et une application en ligne de commandes.

Afin d'avoir un rendu graphique, vous devez télécharger la dernière version de graphiz pour permettre un affichage sous forme d'arbre :

http://www.graphviz.org/Download.php

Vous devez accepter les termes de la licence CPL puis télécharger le sous le bon système d'exploitation.
Pour une facilité d'utilisation vous devez indiquer le répertoire d'installation de dot dans votre PATH (.../graphviz_<version>_/bin sous windows par défaut). Ce n'est normalement pas utile sous linux.

Concernant l'interface web, celle ci exécute du code PHP ; vous devez donc posséder un serveur de type apache en local. (Logiciel rapide d'installation : http://www.easyphp.org/ sous Windows par exemple) Si vous n'en possédez pas ou si vous ne voulez pas en installer vous devrez vous contenter du programme en ligne de commande.

Le programme proprement dit, est compilé avec GCC 4.5 (sous Mingw32 pour Windows)

Les commandes make disponibles sont :

- make all : crée le programme normalement
- make clean : efface les fichiers inutiles
- make test : lance le programme afin d'essayer si il marche, nécessite DOT pour créer le graphique.
- make database : crée le programme : test_database qui affiche les erreurs du fichier source.
- make jaro : crée un programme de test des fonctions Jaro et Jaro-Winkler
- make mktest : crée le programme : gencommandetest qui génére les lignes de commandes permettant créer tout les graphiques correspondants à des changements de paramètres. (cela sert aux tests surtout)

a) fonctionnement de l'interface web.

Pour démarrer l'interface graphique, vous devez ouvrir dans votre navigateur le dossier interface (au préalablement placé dans l'arborescence gérée par appache) que l'on vous a fourni. Vous devez également faire attention à ce que la base de données et l'exécutable soit placés dans le dossier interface/pages.
Le jour de la démonstration nous aurons placé de façon temporaire l'interface sur un serveur http.

Une fois sur la page d'accueil ouverte il ne vous reste plus qu'à cliquer sur la page génération des chemins sous forme graphique. Là il ne vous reste plus qu'a remplir les différents champs, choisir une hauteur maximale, une hauteur minimale et choisir la distance souhaitée.
Attention, si la machine hôte est sous un système Linux, il faut que le '.' soit dans votre variable d'environnement PATH.

b) fonctionnement en mode console

Pour avoir une liste de tous les paramètres tapez "main", "main -h" ou "main -help" dans une console.

L'appel se fait de cette forme :

main -fr:voyage -en:travel -fout:graph_jar_0.dot -hmax:2 -nb:15 -dist:jar -v -sens:0

- -fr:xxx => mot français
- -en:xxx => mot anglais
- -fout:xxx => fichier.dot de sortie
- -hmax:X => hauteur max
- -nb:X => nombre minimum de mots finaux
- -dist:xxx => la distance
  liste de distances :
  lev / jar / jwk / snj / mpj correspondent respectivement à Levenshtein, Jaro, Jaro-Winkler, SoundEx+JaroW, DoubleMétaphone+JaroW.
- -v => mode verbeux : affiche le détail des opérations.
- -sens:X => sens de lecture dans la base de données : 0 sens inverse, 1 classique, 2 les deux sens
- -c:X => la couleur d'affichage du graphique : (0 ou 1)
  0 : dégradé Rouge > Vert
  1 : dégradé Bleu > Vert (plus c'est vert et plus la distance est proche de 1.0)
- -del => Ne supprime pas les données en mémoire (laisse l'OS s'en charger, moins propre mais plus rapide).
- -nothreads => Permet de ne pas utiliser les threads lors du calcul. (peut servir à des tests)

Vous devez ensuite appeler le programme dot.

Exemple :
dot graph_jar_0.dot -Gcharset=latin1 -Tpng -o graph_jar_0.png

Un premier paramètre qui est votre fichier.dot généré par notre programme.

-Gcharset=latin1 correspond à l'encodage des caractères du fichier généré par notre programme.

-Tps indique le format de l'image voulu (-Tpng pour une image png, -Tjpg pour du jpg …)

-o : spécifie le fichier de sortie.

8. Tests

Maintenant que nous avons expliqué le fonctionnement de notre programme, nous allons vous montrer quelques tests de qualités. Nous allons vous montrer quelques chemins obtenus avec les différentes méthodes et sur différents mots (graphe présents dans le dossier graphes testés).

Nous avons attribués à chacun des graphiques une note sur dix en fonction de la qualité estimée des chemins retournés. Ces notes sont totalement subjectives et son indépendantes des distances calculées et présentes sur les graphiques.

En effet les pourcentages de similarités présents sur les graphiques ne sont pas uniformes :

Avoir un pourcentage de similarité de 0,800 avec la distance de Levenshtein ne veut pas signifier la même chose qu'avec la distance de Jaro .

	Ensemble/ together	Voyage/ travel	Roder/ grind	Pouces/ inches	Constitution/ Establishment
lev	6	6	7	4	7
jar	4	5	6	6	7
jwk	5	6	7	7	8
snj	5	6	5	5	4
mpj	4	7	6	4	4

lev / jar / jwk / snj / mpj correspondent respectivement à Levenshtein, Jaro, Jaro-Winkler, SoundEx+Jaro-Winkler, DoubleMétaphone+JaroW.

Au premier abord que la qualité des chemins obtenus est légèrement supérieure avec la distance de Jaro-Winkler. En effet, nous rappelons que celle ci favorise les mots ayant des préfixes communs ce qui permet de retenir plus facilement une association de mots.

Nous avons attribué au distances phonétiques (snj et mpj) des notes légèrement inférieures car celles ci sont de qualités moindre. En fonction de la prononciation du mot anglais à rechercher la qualité peut être assez décevante.

Nous avons également pu constater qu'en fonction des mots anglais à rechercher la qualité des chemins variée.

En effet together et inches ont peu de mot très ressemblant dans la langues française :

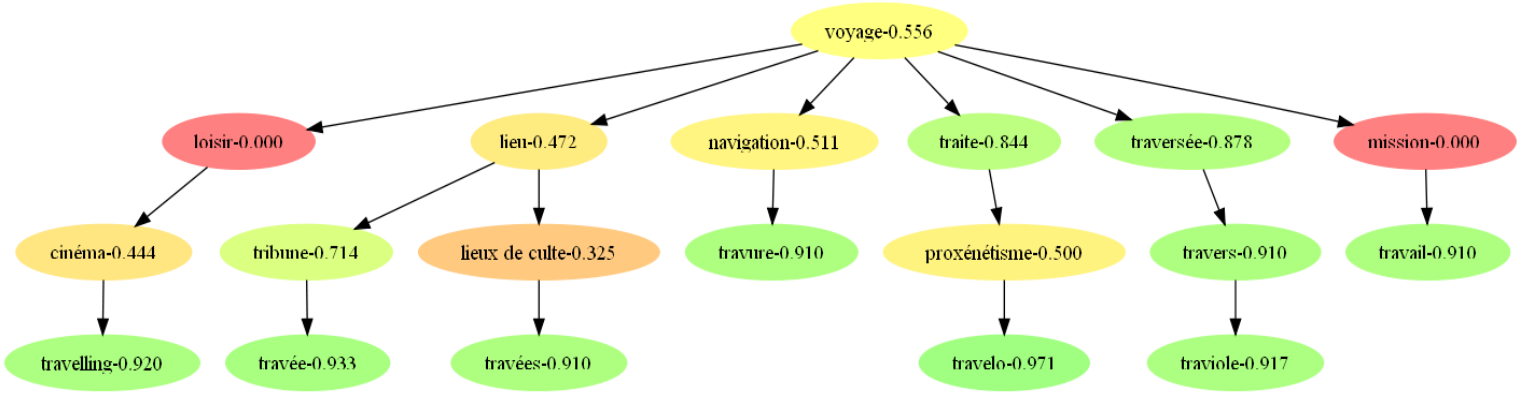
Pour together nous trouvons juste toaster comme mot ayant une petite ressemblance et inciser qui est le mot le plus ressemblant de inches.

Par contre ce n'est pas le cas pour travel, grind et Establishment.

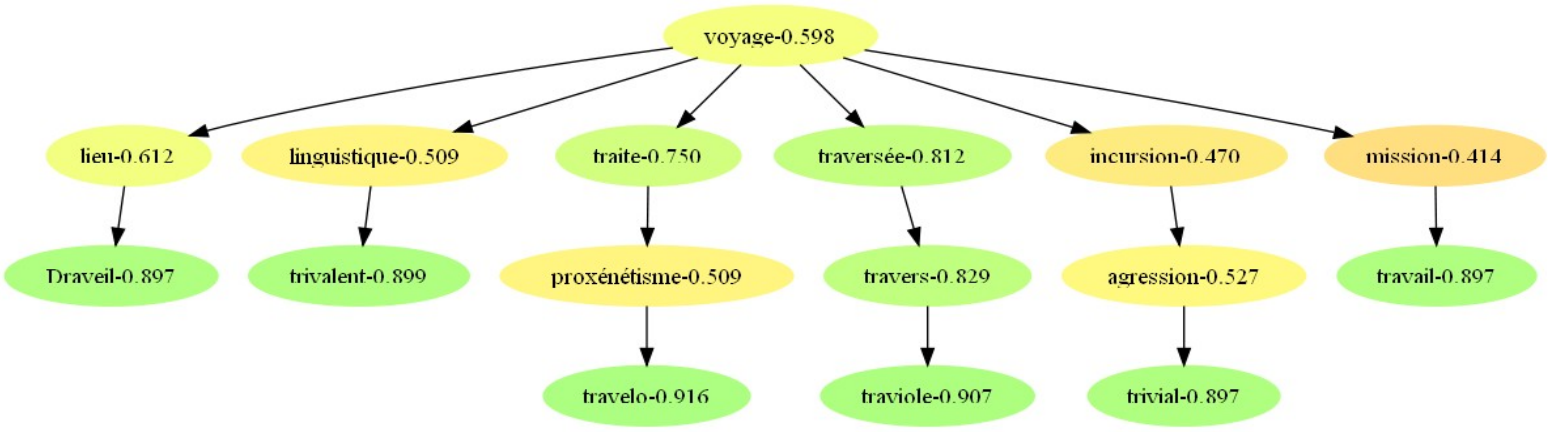
Travel : travelo, travesti, travelling ...
Grind : grand, grain , grincheux, gin..
Establishment : estimable, estamper, embellissement ...

En conclusion, il est difficile d'établir une hiérarchie claire dans les distances (Jaro-Winkler nous semble être légèrement la plus efficace). En effet suivant le couple de mot à associer une méthode peut donner de meilleurs résultats graphiques.

Exemple de graphiques :
couple de mots Voyage, Travel / Jaro-Winkler / 8 mots maxi / hauteur max : 10 / sens normal :



couple de mots Voyage, Travel / Double Métaphone + Jaro-Winkler / 8 mots maxi / hauteur max : 10 / sens normal :



9. Problèmes et perspectives

9.1 Problèmes

Après plusieurs simulations nous nous sommes rapidement rendu compte rendus compte que la base de données devait être adaptée pour correspondre à nos besoins.
Nous avons listé les principaux défauts :

- Elle contient des relations pointant vers des mots inexistants.
- Elle contient des noms non cohérents (ex.:"_FL"), ne correspondant à aucun mots réels.
- Elle contient des noms a peut être prendre en compte (ex:"c4:feu") (voir ligne 26 de la BDD JeuDeMot)
- Elle contient des relations pointant plusieurs fois vers le même eid (en ayant un type et poid différent)
- Elle contient des relations vers le mot d'origine (boucles)
- le type n'est pour l'instant pas significatif et pas assez détaillé
- le poids (fréquence d'apparition) n'est lui non plus pas significatif : de nombreux Nodes ont un poids de 0, pareil pour les relations.

Pour répondre à cela nous avons créé des fonctions de filtrage chargées d'enlever les mots ou relations non utilisable par le programme.
Vous pouvez voir la liste des mots filtrés dans le fichier pasbon.txt
Ce fichier peut être régénéré en exécutant la commande : make database

Le deuxième inconvénient concerne les deux derniers points. La notion d'efficacité est censée prendre le compte le poids et le type de relations. Malheureusement il y a trop d'incohérences : on ne peut pas déduire un calcul d'efficacité quand la majorité des nodes ont un poids de 0 et quand le type de relation est trop aléatoire.
Pour définir la notion d'efficacité, on s'est donc contenter des valeurs que renvoyaient les fonctions de comparaisons de deux mots.

Le programme dans l'état actuel, renvoie un graphique composé de mots uniques. On ne trouves pas deux fois le même mot dans le graphe, or plusieurs chemins de même hauteur peuvent mener à un mot considéré comme un bon résultat. Cela n'es actuellement pas pris en compte pour des soucis de rapidité de traitement.

9.2 Perspectives

Voici quelques améliorations possibles pour divers parties du programme :

La première amélioration possible concerne la prise de contrôle lors de la génération d'un graphique. Comme par exemple pouvoir cliquer sur un mot du graphique généré pour demander plus de chemin utilisant ce mot.
C'est une partie du projet que nous pensions développe comme nous l'avions écris dans la cahier des charges que malheureusement nous avons du abandonner. Malgré cela nous pensons toujours que cela pourrait être intéressant.
Cependant la raison qui nous à obligé à abandonner cette option n'est pas seulement dû au temps mais aussi à un manque de connaissance sur les technologies HTML MAP, JAVASCRIPT et PHP.

On pourrait également développer un système de client serveur avec côté serveur un daemon qui tourne avec la base de données déjà sous forme d'arbre. Cela nous ferait gagner quelques secondes (durée de création de la base de données) pour chaque génération de graphique sur un ordinateur personnel récent, au sinon utiliser les threads pour créer cet arbre. Cela serait vraiment secondaire comme développement car ce

n'est pas ce qui prend le plus de temps lors d'une génération de chemin. Néanmoins ce n'est quand même pas négligeable pour le confort de l'utilisateur.

Cette amélioration nécessiterait néanmoins une adaptation de l'interface graphique côté client qui devra envoyer les demandes de l'utilisateur au serveur.

Côté serveur il y a plusieurs possibilités. Soit la génération de l'image du graphique se fait côté serveur. Le logiciel client se contenterait alors d'afficher l'image.

Soit le serveur envoie la liste des chemins au client, et c'est le logiciel client qui est alors chargé de traiter les données.

Une amélioration essentielle serait de travailler sur une nouvelle base de données. On pourra prendre comme base propre une base de données wiki.

On pourrait essayer de gagner quelques secondes lors de la création des chemins. En effet dès que l'on demande un graphique un peu complet le temps de création dépasse rapidement les dix secondes.

On pourrait également essayer d'améliorer la définition de l'efficacité. Actuellement l'efficacité ne prend juste en compte que la similarité avec le mot rechercher.

On pourrait introduire d'autres paramètres :

–La hauteur d'un chemin (plus un chemin est long moins il est facile de se souvenir de tous les mots).

–La similarité entre les mots du chemin (plus les mots d'un même chemin ont une association forte, plus il sera facile de se souvenir d'un nombre important de mots).

–Poids du mot (plus un mot est fréquemment utilisée dans la langue française, plus il sera facile de s'en souvenir).

–Type du mot (une relation peut être plus efficace à retenir en fonction de son type).

Il serait bien aussi de tester des distances basées sur des études linguistiques plus poussées. (Étymologie etc...)

10. Conclusions

10.1 du groupe de travail

La répartition des tâches entre les membres du groupes nous a permis d'avancer efficacement ; chacun travaillait de son côté tout en gardant contact avec les autres via internet pour se tenir au courant de l'avancement du projet, puis nous réunissions lors de réunions hebdomadaires puis espacées de trois à quatre semaines, le plus souvent avec notre tuteur. Lors de ces réunions nous faisions le point sur l'avancement du projet, en cas de désaccord nous débattions sur la ou les meilleures solutions, puis nous fixions des objectifs pour la réunion suivante.

Nous sommes issue de plusieurs formations, et bénéficions d'option différentes, grâce à cela nous avons pu plus facilement nous entraider lorsque l'un de nous n'arriver pas à avancer sur sa partie.

Nous aurions souhaité un peu plus de temps afin de pouvoir pourvoir notre programme de certaines fonctionnalités comme une certaine interactivité entre l'utilisateur et le graphe, afin qu'il détermine un chemin qui lui convienne, toutefois, l'application est malgré tout opérationnelle.

10.2 de l’application

Actuellement, nous n'avons pas trouvé de bogue dans la version finale que nous vous montrons. Nous avons testé avec GCC 4.5 sous Mingw32 pour Windows, et GCC 4.4 sous Unix (32 bits et 64 bits). L'outil de génération de graphe, dot, fonctionne très simplement et génère des graphes clairs à partir des données que l'application lui fournit. Toutefois, en ce qui concerne la base de données, il aurait peut être été intéressant d'en avoir une moins bruitée, et surtout plus complète, afin d'obtenir des chemins plus efficace. De plus faire tourner l'application côté serveur permet un certain gain de temps pour l'utilisateur.

Au niveau de l'analyse, nous avons un peu négligé la création des chemins et de ce fait, il nous a parfois fallu prendre des décisions rapidement sans savoir ce que cela aurait comme résultat.

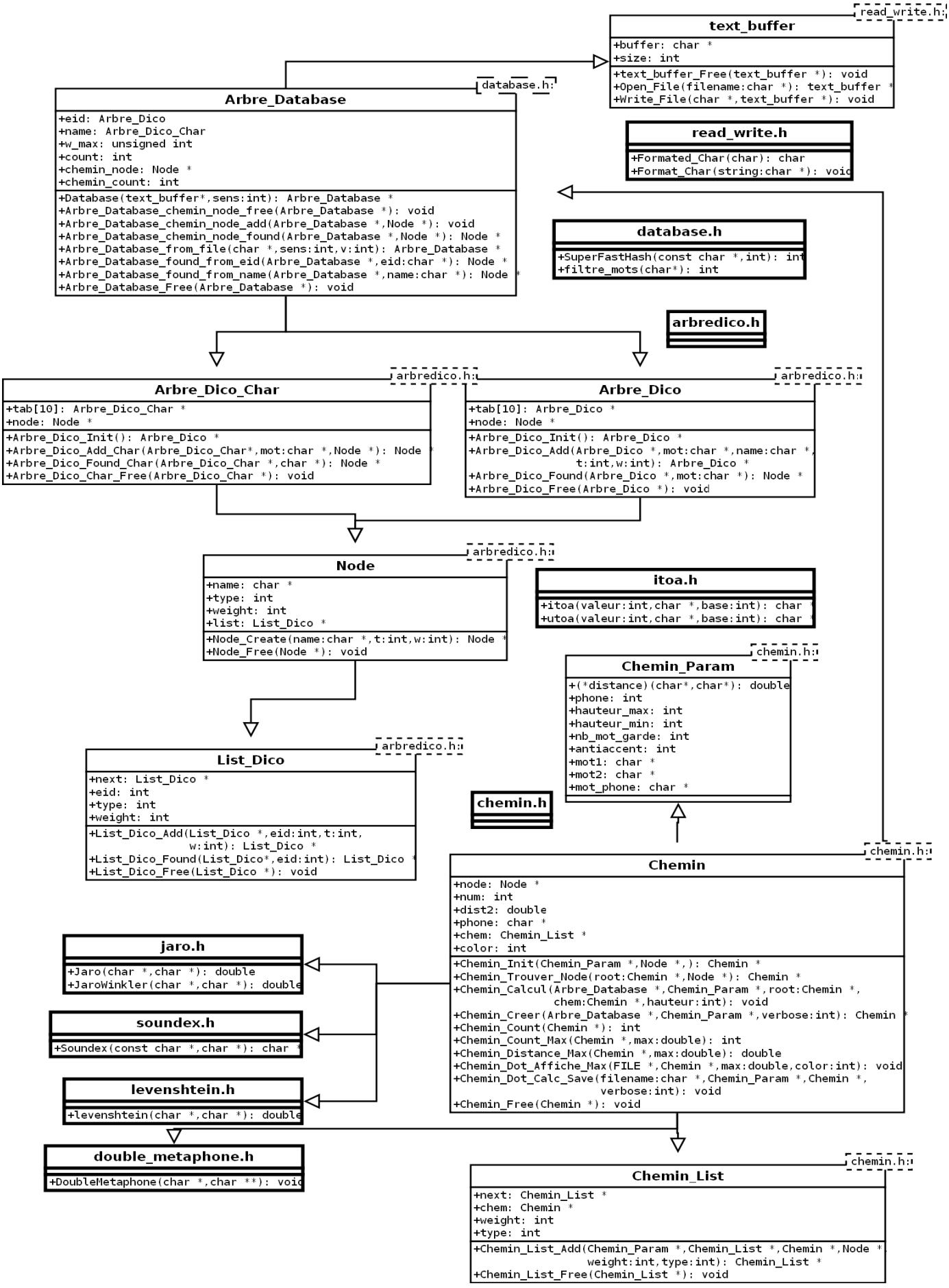
Au final la génération propose quelque fois des chemins mnémotechniques intéressants mais l'efficacité de ce moyen mnémotechnique peut être remise en question. Pour certains mots : peu, voir aucun chemin au final nous semblent vraiment être de bonnes solutions au problème par rapport à la quantité de chemins alternatifs.

Ce programme reste donc plus à but pédagogique que pour une utilisation au quotidien.

11. Annexe A : documents d'analyse

Documentation via des graphiques résumés :

Fichier	Description rapide
arbredico.c / .h	• Fonctions servant à mémoriser un dictionaire sous forme d'arbre • Définitons d'un Node • Fonctions utiles sur les listes de relations
changes.txt	• Informations concernant l'avancée du projet
chemin.c / .h	• Fonctions servant à créer, trier les chemins • Définitions des chemins, et des paramètres utilisateurs à faire passer
database.c / .h	• fonctions servant à transformer et utiliser la base de donnée de JeuDeMots en dico. sous forme d'arbre • filtre permettant de ne pas garder les mots "bruités" de la base de donnée JeuDeMots • fontion de hashage (peut être réutilisée autre part)
double_metaphone.c / .h	• Distance Double-Métaphone (distance phonétique)
gencommandetest.c	• Permet de tester la base de donnée (crée le fichier pasbon.txt)
itoa.c	• Fonctions plus rapides, permettant d'éviter des sprintf
jaro.c /.h	• Fonction de calcul de la distance de Jaro et Jaro-Winkler
jaro_test.c	• Fichier permettant de tester rapidement une distance entre 2 mots
levenshtein.c / .h	• Distance de Levenshtein
main.c	• Contient un exemple d'utilisation des fonctions de génération de dico. sous forme d'arbre a partir de la BDD de JeuDeMots
makefile	MakeFile des fichiers tests : • make clean : ménage • make all : génére un binaire permettant de tester le main.c • make jaro_test : génére un binaire permettant de tester jaro_test.c
pasbon.txt.gz	• Fichier compressé contenant un listing des erreurs trouvées lors du traitement du fichier source de JeuDeMots.
read_write.c / .h	• Fonctions servant à stocker en Ram un fichier pour un traitement rapide par la suite • Fonctions servant à écrire sur le disque le fichier traité (ram to disk)
soundex.c / .h	• Distance SoundEx (distance phonétique)


read_write.h
text_buffer
+buffer: char *
+size: int
+text_buffer_Free(text_buffer *): void
+Open_File(filename:char *): text_buffer *
+Write_File(char *,text_buffer *): void
database.h
Arbre_Database
+eid: Arbre_Dico
+name: Arbre_Dico_Char
+w_max: unsigned int
+count: int
+chemin_node: Node *
+chemin_count: int
+Database(text_buffer*,sens:int): Arbre_Database *
+Arbre_Database_chemin_node_free(Arbre_Database *): void
+Arbre_Database_chemin_node_add(Arbre_Database *,Node *): void
+Arbre_Database_chemin_node_found(Arbre_Database *,Node *): Node *
+Arbre_Database_from_file(char *,sens:int,v:int): Arbre_Database *
+Arbre_Database_found_from_eid(Arbre_Database *,eid:char *): Node *
+Arbre_Database_found_from_name(Arbre_Database *,name:char *): Node *
+Arbre_Database_Free(Arbre_Database *): void
read_write.h
+Formated_Char(char): char
+Format_Char(string:char *): void
database.h
+SuperFastHash(const char *,int): int
+filtre_mots(char*): int
arbredico.h
arbredico.h
Arbre_Dico_Char
+tab[10]: Arbre_Dico_Char *
+node: Node *
+Arbre_Dico_Init(): Arbre_Dico *
+Arbre_Dico_Add_Char(Arbre_Dico_Char*,mot:char *,Node *): Node *
+Arbre_Dico_Found_Char(Arbre_Dico_Char *,char *): Node *
+Arbre_Dico_Char_Free(Arbre_Dico_Char *): void
arbredico.h
Arbre_Dico
+tab[10]: Arbre_Dico *
+node: Node *
+Arbre_Dico_Init(): Arbre_Dico *
+Arbre_Dico_Add(Arbre_Dico *,mot:char *,name:char *,
t:int,w:int): Arbre_Dico *
+Arbre_Dico_Found(Arbre_Dico *,mot:char *): Node *
+Arbre_Dico_Free(Arbre_Dico *): void
arbredico.h
Node
+name: char *
+type: int
+weight: int
+list: List_Dico *
+Node_Create(name:char *,t:int,w:int): Node *
+Node_Free(Node *): void
itoa.h
+itoa(valeur:int,char *,base:int): char *
+utoa(valeur:int,char *,base:int): char *
chemin.h
Chemin_Param
+(*distance)(char*,char*): double
+phone: int
+hauteur_max: int
+hauteur_min: int
+nb_mot_garde: int
+antiaccent: int
+mot1: char *
+mot2: char *
+mot_phone: char *
arbredico.h
List_Dico
+next: List_Dico *
+eid: int
+type: int
+weight: int
+List_Dico_Add(List_Dico *,eid:int,t:int,
w:int): List_Dico *
+List_Dico_Found(List_Dico*,eid:int): List_Dico *
+List_Dico_Free(List_Dico *): void
chemin.h
chemin.h
Chemin
+node: Node *
+num: int
+dist2: double
+phone: char *
+chem: Chemin_List *
+color: int
+Chemin_Init(Chemin_Param *,Node *,): Chemin *
+Chemin_Trouver_Node(root:Chemin *,Node *): Chemin *
+Chemin_Calcul(Arbre_Database *,Chemin_Param *,root:Chemin *,
chem:Chemin *,hauteur:int): void
+Chemin_Creer(Arbre_Database *,Chemin_Param *,verbose:int): Chemin *
+Chemin_Count(Chemin *): int
+Chemin_Count_Max(Chemin *,max:double): int
+Chemin_Distance_Max(Chemin *,max:double): double
+Chemin_Dot_Affiche_Max(FILE *,Chemin *,max:double,color:int): void
+Chemin_Dot_Calc_Save(filename:char *,Chemin_Param *,Chemin *,
verbose:int): void
+Chemin_Free(Chemin *): void
jaro.h
+Jaro(char *,char *): double
+JaroWinkler(char *,char *): double
soundex.h
+Soundex(const char *,char *): char *
levenshtein.h
+levenshtein(char *,char *): double
double_metaphone.h
+DoubleMetaphone(char *,char **): void
chemin.h
Chemin_List
+next: Chemin_List *
+chem: Chemin *
+weight: int
+type: int
+Chemin_List_Add(Chemin_Param *,Chemin_List *,Chemin *,Node *,
weight:int,type:int): Chemin_List *
+Chemin_List_Free(Chemin_List *): void

database.h:

Arbre_Database

+eid: Arbre_Dico
+name: Arbre_Dico_Char
+w_max: unsigned int
Poid max que peut avoir un Node

+count: int
contient le nombre de mots (de Nodes) de l'arbre

+chemin_node: Node *
Node[count] utilisé pour la création des chemins, ce tableau contient la liste des Nodes déjà présent dans lors de la création des Chemins.

+chemin_count: int

+Database(text_buffer*,sens:int): Arbre_Database *
Crée l'Arbre_Database permettant d'obtenir le Node (et donc liste de relation) a partir d'un nom ou d'un eid.
sens = 0 relations dans le sens normal, sens = 1 les relations sont inversées, sens = 2 les relations sont ajoutées dans les 2 sens

+Arbre_Database_chemin_node_free(Arbre_Database *): void
Sert à la création des chemins

+Arbre_Database_chemin_node_add(Arbre_Database *,Node *): void
Sert à la création des chemins

+Arbre_Database_chemin_node_found(Arbre_Database *,Node *): Node *
Sert à la création des chemins

+Arbre_Database_from_file(char *,sens:int,v:int): Arbre_Database *
Fonction permetant de renvoier directement un Arbre_Database à partir d'un fichier de base de donnée de JeuDeMots
sens = 0 relations dans le sens normal, sens = 1 les relations sont inversées, sens = 2 les relations sont ajoutées dans les 2 sens.
v = verbose affiche ou pas les étapes de fabrication

+Arbre_Database_found_from_eid(Arbre_Database *,eid:char *): Node *
Récupére un Node a partir d'un eid et d'un Arbre_Database déjà créé

+Arbre_Database_found_from_name(Arbre_Database *,name:char *): Node *
Récupére un Node a partir d'un mot et d'un Arbre_Database déjà créé

+Arbre_Database_Free(Arbre_Database *): void
Libération mémoire d'un Arbre_Database

itoa.h
fonctions permettant d'éviter d'utiliser sprintf

+itoa(valeur:int,char *,base:int): char *
Convertit un entier en chaine de caractères selon une base donnée (de manière plus rapide qu'un sprintf)

+utoa(valeur:int,char *,base:int): char *
Convertit un entier non signé en chaine de caractères selon une base donnée (de manière plus rapide qu'un sprintf)

database.h

+SuperFastHash(const char *,int): int
Permet de simplifier l'arbre nécessaire pour pouvoir retrouver un Node à partir d'une chaine de caractères.

+filtre_mots(char*): int
Permet de filtrer les mots qui seront sauvegardées (et utilisés) dans l'Arbre_Database. Certains mots contenant des '_' ou des '+' sont du bruit et sont inutilisables il faut donc les filtrés afin de ne pas avoir d'erreur.
Renvoie 0 si le mot n'est pas désiré sinon renvoie 1

read_write.h:

text_buffer

+buffer: char *
+size: int

+text_buffer_Free(text_buffer *): void
Libération mémoire d'un text_buffer

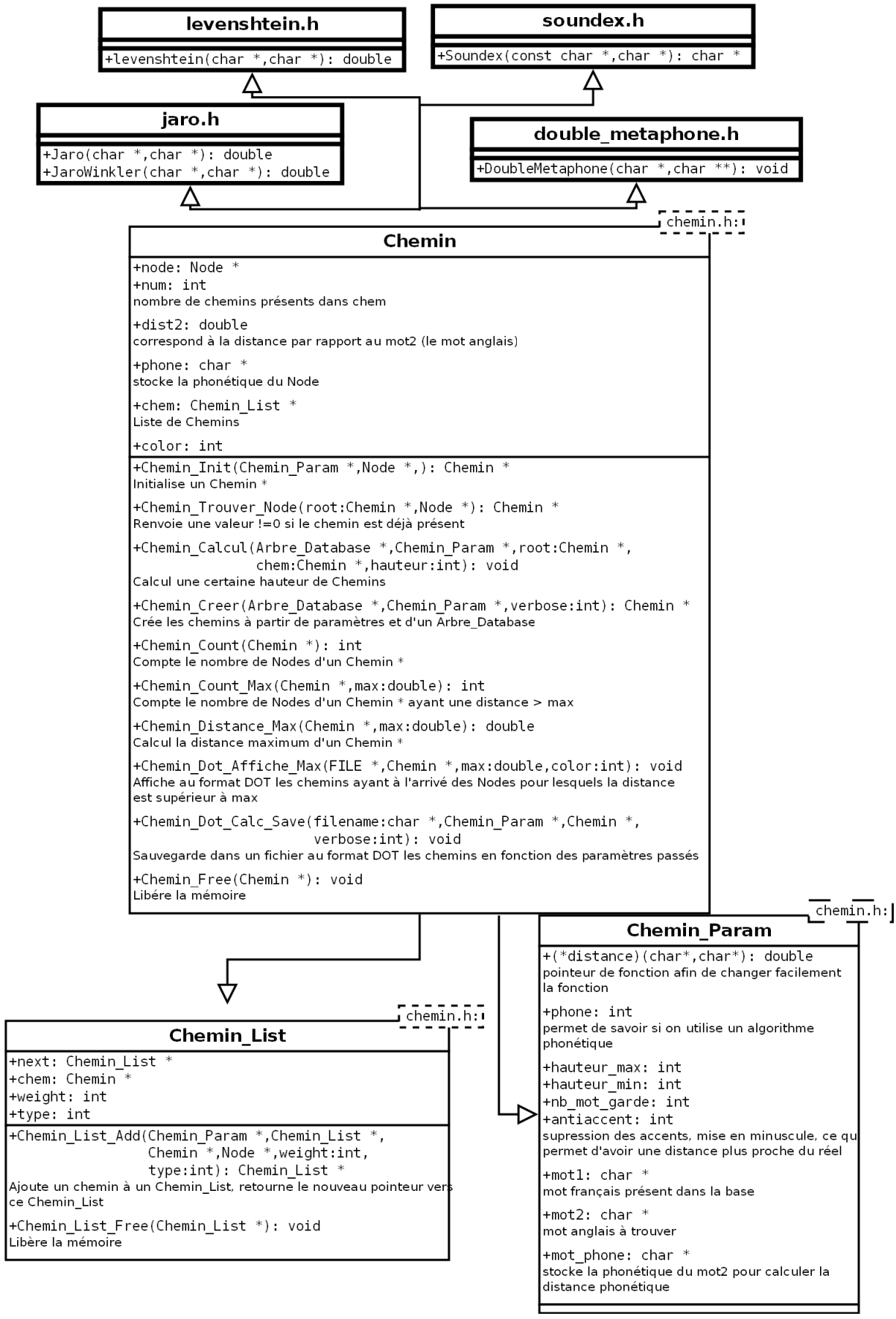
+Open_File(filename:char *): text_buffer *
Crée un text_buffer contenant le fichier 'filename' ou renvoie 0

+Write_File(char *,text_buffer *): void

read_write.h

+Formated_Char(char): char
Renvoie le caractère modifié (sans accent, en minuscule) si il est autorisé, sinon renvoie 0

+Format_Char(string:char *): void
Modifie la chaine de caractères : enlève les accents et met tout en minuscule, enlève les caractères non autorisés.


levenshtein.h
+levenshtein(char *,char *): double
soundex.h
+Soundex(const char *,char *): char *
jaro.h
+Jaro(char *,char *): double
+JaroWinkler(char *,char *): double
double_metaphone.h
+DoubleMetaphone(char *,char **): void
chemin.h:
Chemin
+node: Node *
+num: int
nombre de chemins présents dans chem
+dist2: double
correspond à la distance par rapport au mot2 (le mot anglais)
+phone: char *
stocke la phonétique du Node
+chem: Chemin_List *
Liste de Chemins
+color: int
+Chemin_Init(Chemin_Param *,Node *,): Chemin *
Initialise un Chemin *
+Chemin_Trouver_Node(root:Chemin *,Node *): Chemin *
Renvoie une valeur !=0 si le chemin est déjà présent
+Chemin_Calcul(Arbre_Database *,Chemin_Param *,root:Chemin *,
chem:Chemin *,hauteur:int): void
Calcul une certaine hauteur de Chemins
+Chemin_Creer(Arbre_Database *,Chemin_Param *,verbose:int): Chemin *
Crée les chemins à partir de paramètres et d'un Arbre_Database
+Chemin_Count(Chemin *): int
Compte le nombre de Nodes d'un Chemin *
+Chemin_Count_Max(Chemin *,max:double): int
Compte le nombre de Nodes d'un Chemin * ayant une distance > max
+Chemin_Distance_Max(Chemin *,max:double): double
Calcul la distance maximum d'un Chemin *
+Chemin_Dot_Affiche_Max(FILE *,Chemin *,max:double,color:int): void
Affiche au format DOT les chemins ayant à l'arrivé des Nodes pour lesquels la distance est supérieur à max
+Chemin_Dot_Calc_Save(filename:char *,Chemin_Param *,Chemin *,
verbose:int): void
Sauvegarde dans un fichier au format DOT les chemins en fonction des paramètres passés
+Chemin_Free(Chemin *): void
Libére la mémoire
chemin.h:
Chemin_Param
+(*distance)(char*,char*): double
pointeur de fonction afin de changer facilement la fonction
+phone: int
permet de savoir si on utilise un algorithme phonétique
+hauteur_max: int
+hauteur_min: int
+nb_mot_garde: int
+antiaccent: int
supression des accents, mise en minuscule, ce qui permet d'avoir une distance plus proche du réel
+mot1: char *
mot français présent dans la base
+mot2: char *
mot anglais à trouver
+mot_phone: char *
stocke la phonétique du mot2 pour calculer la distance phonétique
chemin.h:
Chemin_List
+next: Chemin_List *
+chem: Chemin *
+weight: int
+type: int
+Chemin_List_Add(Chemin_Param *,Chemin_List *,
Chemin *,Node *,weight:int,
type:int): Chemin_List *
Ajoute un chemin à un Chemin_List, retourne le nouveau pointeur vers ce Chemin_List
+Chemin_List_Free(Chemin_List *): void
Libère la mémoire

12. Annexe B : Listings identifiés et commentés

Listing des fichiers d'en-tête :

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : arbredico.h
Date : 09/03/2010
Description :
  fonctions servant à mémoriser un dictionaire sous forme d'arbre
  définitons d'un Node
  fonctions utiles sur les listes de relations
*/

#ifndef arbredico_h
#define arbredico_h

  typedef struct Arbre_Dico {
    struct Arbre_Dico* tab[10];
    struct Node *node;
  } Arbre_Dico;
  //Structure Arbre_Dico
  //Si node = 0 cela veut dire qu'il n'y a pas de node associé, donc l'eid
n'est pas fini

  typedef struct Arbre_Dico_Char {
    struct Arbre_Dico_Char* tab[10];
    struct Node *node;//char *eid;
  } Arbre_Dico_Char;
  //Structure Arbre_Dico_Char
  //Si eid = 0 cela veut dire qu'il n'y a pas de node associé, donc le mot
n'est pas fini

  typedef struct List_Dico {
    struct List_Dico* next;
    int e;
    int t;
    int w;
  } List_Dico;
  //Structure List_Dico
  //e est le eid du mot
  //t est le type
  //w est le poids (weight)

  typedef struct Node {
    char *name;
    int t;
    int w;
    struct List_Dico* list;
  } Node;
  //Structure Node
  //name est le mot (en ascii)
  //t est le type
  //w est le poids (weight)
  //list contient une List_Dico des relations liées à ce mot

  //Fonctions sur les List_Dico
```

```
  List_Dico* List_Dico_Add(List_Dico* t2,int eid,int t,int w);
  List_Dico* List_Dico_Found(List_Dico* li,int eid);
  void List_Dico_Free(List_Dico* t2);

  //Fonctions sur les Node
  Node* Node_Create(char *name,int t,int w);
  void Node_Free(Node* n);

  //Fonctions sur les Arbre_Dico et Arbre_Dico_Char
  Arbre_Dico* Arbre_Dico_Init();
  Arbre_Dico* Arbre_Dico_Add(Arbre_Dico* tt,char *mot,char* name,int t,int
w);
  Node * Arbre_Dico_Add_Char(Arbre_Dico_Char* tt,char *mot,Node *n);
  Node* Arbre_Dico_Found(Arbre_Dico* t,char *mot);
  Node* Arbre_Dico_Found_Char(Arbre_Dico_Char* t,char *mot);
  void Arbre_Dico_Free(Arbre_Dico* t);
  void Arbre_Dico_Char_Free(Arbre_Dico_Char* t);

#endif

//Fin de fichier
```

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : main.c
Date : 26/03/2010
Description : contient les fonctions sur la création, gestion des Chemins à
partir des données utilisateurs)
*/

#ifndef chemin_h
#define chemin_h

#ifndef max
#define max(x,y) ((x)>(y)?(x):(y))
#endif

  typedef struct Chemin_Param {
    double (*distance)(char*,char*); //fonction de distance prend en
compte 2 char* et renvoie un double compris entre 0 et 1
    int phone;
    int hauteur_max;
    int hauteur_min; //Peut servir par la suite...
    int nb_mot_garde;
    int antiaccent; //Enlève les accents avant d'appliquer un algo de
distance
    char *mot1; //Mot Français
    char *mot2; //Mot Anglais
    char *mot_phone; //Mot en Phonétique
    int color;
  } Chemin_Param;

  typedef struct Chemin_List {
    struct Chemin_List *next;
    struct Chemin *chem;
    //double dist; //Distance entre 2 mots du chemin, ne sert pas pour
```

```
l'instant
     int w;
     int t;
  } Chemin_List;

  typedef struct Chemin {
     Node* node;
     int num; //Coute pas si cher en mémoire et évite de parcourir la liste
pour les calculs
     //double dist1; //Distance par rapport au mot d'origine, ne sert pas
pour l'instant
     double dist2; //Distance par rapport au mot d'arrivé
     char *phone; //Distance Phonétique
     struct Chemin_List* chem;
  } Chemin;

  typedef struct Thread_Param {
     Arbre_Database *t;
     Chemin_Param *cp;
     Chemin_List *cl;
     Chemin *root;
     int hauteur;
     int nb;
  } Thread_Param;

  double distance(Chemin_Param *cp,char *st1,char *st2);
  //Calcul la distance entre 2 mots en fonctions des paramètres passés.

  //Fontions sur les chemins

  Chemin *Chemin_Init(Chemin_Param *cp,Node *mot);
  //Initialise un Chemin *
  Chemin *Chemin_Trouver_Node(Chemin *root,Node *no);
  //Renvoie une valeur !=0 si le chemin est déjà présent
  void Chemin_Calcul(Arbre_Database *t,Chemin_Param *cp,Chemin *chem,Chemin
*root,int hauteur);
  //Calcul une certaine hauteur de Chemins
  void Chemin_Calcul_thread(Arbre_Database *t,Chemin_Param *cp,Chemin
*chem,Chemin *root,int hauteur,int nb_threads);
  //Calcul une certaine hauteur de Chemins avec la gestion de threads
  Chemin *Chemin_Creer(Arbre_Database *t,Chemin_Param *cp,int v,int
nb_threads);
  //Crée les chemins à partir de paramètres et d'un Arbre_Database
  int Chemin_Count(Chemin *root);
  //Compte le nombre de Nodes d'un Chemin *
  int Chemin_Count_Max(Chemin *root,double max);
  //Compte le nombre de Nodes d'un Chemin * ayant une distance > max
  double Chemin_Distance_Max(Chemin *root,double max);
  //Calcul la distance maximum d'un Chemin *
  void Chemin_Dot_Affiche_Max(FILE *f,Chemin *root,double max,int color);
  //Affiche au format DOT les chemins ayant à l'arrivé des Nodes pour
lesquels la distance est supérieur à max
  void Chemin_Dot_Calc_Save(char *filename,Chemin_Param *cp,Chemin
*root,int v);
  //Sauvegarde dans un fichier au format DOT les chemins en fonction des
paramètres passés
  void Chemin_Free(Chemin *root);
  //Libére la mémoire

  Chemin_List *Chemin_List_Add(Chemin_Param *cp,Chemin_List *cl,Chemin
*chem,Node *nroot,int w,int t);
```

```
  //Ajoute un chemin à un Chemin_List, retourne le nouveau pointeur vers ce
Chemin_List
  void Chemin_List_Free(Chemin_List *cl);
  //Libére la mémoire

  void strcouleur(int color,double dist,char *st);

  // void Chemin_Affiche_Dot(FILE *f,Chemin *root);
  // void Chemin_Dot_Save(char *filename,Chemin *ch);
  //Ces fonctions servent à sauvegarder le graph en entier, elles ne sont
pas utilisées au final

#endif
```

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : database.h
Date : 09/03/2010
Description : fonctions servant à transformer et utiliser la base de donnée
de JeuDeMots en dico. sous forme d'arbre
  filtre permettant de ne pas garder les mots "bruités" de la base de
donnée JeuDeMots
  fontion de hashage (peut être réutilisée autre part)
*/

#ifndef database_h
#define database_h

  typedef struct Arbre_Database {
    struct Arbre_Dico* eid;
    struct Arbre_Dico_Char* name;
    unsigned int w_max;
    int count;
    Node *chemin_node; //Tableau contenant les nodes déjà présent dans
l'arbre des chemins, parcourir en O(n) ce tableau est largement plus rapide
que de chercher à chaque fois en récursif dans l'arbre des chemins
    int chemin_count;
  } Arbre_Database;
  //Structure Arbre_Database
  //contient un Arbre_Dico donnant le node à partir d'un eid
  // et un Arbre_Dico donnant l'eid à partir d'un mot
  // Pour le second dico, l'eid est stocké comme un char * (à la place d'un
node)

  int SuperFastHash(const char * data, int len);
  //Fonction de Hashage des noms afin de simplifier et normaliser les noms
dans l'Arbre_Database, permet par la même occasion de prendre moins de
place en mémoire
  // http://www.azillionmonkeys.com/qed/hash.html

  int filtre_mots(char* st);
  //Permet de filtrer les mots qui seront sauvegardées (et utilisés) dans
l'Arbre_Database
  //Certains mots contenant des '_' ou des '+' sont du bruit et sont
inutilisables il faut donc les filtrés afin de ne pas avoir d'erreur
  //renvoie 0 si le mot n'est pas désiré sinon renvoie 1
```

```
  Arbre_Database* Database(text_buffer* txt,int sens);
  //Crée l'Arbre_Database permettant d'obtenir le Node (et donc liste de
relation) a partir d'un nom ou d'un eid en un temps très court
  //sens = 0 relations dans le sens normal, sens = 1 les relations sont
inversées, sens = 2 les relations sont ajoutées dans les 2 sens.

  Arbre_Database *Arbre_Database_from_file(char *file,int sens,int v);
  //Fonction permetant de renvoier directement un Arbre_Database à partir
d'un fichier de base de donnée de JeuDeMots
  //sens = 0 relations dans le sens normal, sens = 1 les relations sont
inversées, sens = 2 les relations sont ajoutées dans les 2 sens.
  //v = verbose affiche ou pas les étapes de fabrication

  Node *Arbre_Database_found_from_eid(Arbre_Database *t,char *eid);
  //Récupére un Node a partir d'un eid et d'un Arbre_Database déjà créé

  Node *Arbre_Database_found_from_name(Arbre_Database *t,char *name);
  //Récupére un Node a partir d'un mot et d'un Arbre_Database déjà créé

  void Arbre_Database_Free(Arbre_Database *a);
  //Libération mémoire d'un Arbre_Database

  //Les fonctions suivantes sont utilisées lors de la création des chemins
afin de stocker les chemins déjà utilisés
  void Arbre_Database_chemin_node_add(Arbre_Database* t,Node *n);
  Node* Arbre_Database_chemin_node_found(Arbre_Database* t,Node *n);
  void Arbre_Database_chemin_node_free(Arbre_Database* t);

#endif

//Fin de fichier
```

```
/*
Licence GPL v3
Auteur : http://search.cpan.org/~maurice/Text-DoubleMetaphone-0.07/ /
Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : double_metaphone.h
Date : 09/03/2010
Description : implémentation de l'algorithme double_metaphone adapté au
projet
http://en.wikipedia.org/wiki/Double_Metaphone
http://en.wikipedia.org/wiki/Metaphone
*/

#ifndef DOUBLE_METAPHONE__H
#define DOUBLE_METAPHONE__H


typedef struct
{
    char *str;
    int length;
    int bufsize;
    int free_string_on_destroy;
}
metastring;
```

```
void DoubleMetaphone(char *str,char **codes);


#endif
```

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : itoa.h
Date : 10/05/2010
Description : integer to ascii, unsigned int to ascii
*/

#ifndef itoa_h
#define itoa_h

char* itoa(int value, char* result, int base);
//Convertit un entier en chaine de caractères selon une base donnée (de
manière plus rapide qu'un sprintf)

char* utoa(int value, char* result, unsigned int base);
//Convertit un entier non signé en chaine de caractères selon une base
donnée (de manière plus rapide qu'un sprintf)

#endif
```

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : jaro.h
Date : 09/03/2010
Description : fonctions de calcul de la distance de Jaro et Jaro-Winkler
http://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance
*/

#ifndef JARO
#define JARO

#define max(x,y) ((x)>(y)?(x):(y))
#define min(x,y) ((x)<(y)?(x):(y))
//Macros min max

#define JARO_CALC_MACRO ((compteMatching-
compteTransposition/2.0)*l1*l2+compteMatching*compteMatching*(l2+l1))/
(compteMatching*l1*l2*3.0);
//Permet de modifier la formule au cas ou pour des tests
#define WINKLER_prefixe 0.1
//Permet de changer le poid du préfixe 0.1 par défault

double Jaro(char *t1,char *t2);
```

```
//Distance de Jaro, renvoie un double compris entre 0 et 1 correspondant à
la distance
double JaroWinkler(char *t1,char *t2);
//Distance de Jaro-Winkler, renvoie un double compris entre 0 et 1
correspondant à la distance

#endif

//Fin de fichier
```

```
/*
Licence GPL v3
Auteur : Berthelot Victor
Projet : Chemin Mnémotechnique
Fichier : levenshtein.h
Date : 13/04/2010
Description : fonction de mesure similarité entre 2 chaines de caractères
basé sur la distance de Levenshtein
http://en.wikipedia.org/wiki/Levenshtein_distance
*/
#ifndef Levenshtein
#define Levenshtein

#define min(x,y) ((x)<(y)?(x):(y))
#define minimum(a,b,c) min(a,min(b,c))

double levenshtein(char *s,char*t);

#endif
```

```
/*
Licence GPL v3
Auteur : Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : read_write.h
Date : 09/03/2010
Description : Sert a stocker un fichier texte en RAM, l'utiliser facilement
par la suite
  Permet aussi d'écrire dans un fichier un texte que l'on viens de traiter
(en mémoire)
*/

#ifndef read_write_h
#define read_write_h


#define max_buffer 1024*1024

typedef struct text_buffer {
  char *buffer;
  int size;
} text_buffer;

void text_buffer_Free(text_buffer *txt);
//Libére en mémoire un text_buffer

text_buffer* Open_File(char *filename);
```

```
/*
Permet de lire un fichier et le mettre en mémoire (RAM)
le text_buffer est en faite un char * contenant un texte et un int
contenant la taille du texte (max 2Go car entier signés)
l'int est utilisé car la base de donnée future ne devrais pas dépasser 1Go
et pour éviter des erreurs de convertions (entre unsiged int et int)
*/

void Write_File(char *filename,text_buffer* txt);
/*
Permet d'écrire le fichier "filename" contenant le text_buffer txt
*/

char Formated_Char(char c);
//Renvoie le caractère modifié (sans accent, en minuscule) si il est
autorisé, sinon renvoie 0
void Format_Char(char *st);
//Modifie la chaine de caractères : enlève les accents et met tout en
minuscule, enlève les caractères non autorisés.

#endif

//Fin de fichier
```

```
/*
Licence GPL v3
Auteur : http://rosettacode.org/wiki/Soundex / Bontemps Jean-Charles
Projet : Chemin Mnémotechnique
Fichier : soundex.h
Date : 10/05/2010
Description :
  implémentation de l'algorithme phonétique Soundex
  http://en.wikipedia.org/wiki/Soundex
  http://rosettacode.org/wiki/Soundex
*/

#ifndef Soundex_h
#define Soundex_h

char *Soundex(const char *word, char *bufr);
//Calcul la distance SoundEx du mot 'word' et la met dans 'bufr'

#endif
```

13. Annexe C : Bibliographie

JeuDeMots:
http://www.lirmm.fr/jeuxdemots/ (Mathieu Lafourcade)
Base de donnée : http://www.lirmm.fr/~lafourcade/JDM-LEXICALNET-FR/

Wikipedia :
http://en.wikipedia.org/wiki/Levenshtein_distance
http://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance
http://en.wikipedia.org/wiki/Soundex
http://en.wikipedia.org/wiki/Metaphone
http://en.wikipedia.org/wiki/Double_Metaphone

SoundEx:
http://rosettacode.org/wiki/Soundex

Double-Métaphone :
http://search.cpan.org/~maurice/Text-DoubleMetaphone-0.07/

L'expérimentation et la démarche scientifique :
http://pagesperso-orange.fr/andre.tricot/TricotNEQDefinitif.pdf