Master Informatique

Rapport du projet d'initiation à la recherche

sujet : Constitution d'une base de cas de corrections du français

Année 2018-2019

Ètudiants : Alex Ginestra
Christopher Klein

Équipe : K et Sémagramme
Encadrants : Bruno Guillaume, Yves Lepage, Jean Lieber et Emmanuel Nauer

Décharge de responsabilité

L'Université de Lorraine n'entend donner ni approbation ni improbation aux opinions émises dans ce rapport, ces opinions devant être considérées comme propres à leur auteur.

Remerciements

Tout d'abord, nous tenons à remercier toute l'équipe pédagogique du département informatique de la faculté des sciences et technologies pour les quatre années de formation en Master informatique.
De plus, nous remercions toutes les personnes qui ont contribué au bon déroulement de notre travail et qui nous ont aidé lors de la rédaction de ce rapport. Premièrement, nous adressons nos remerciements à notre professeur, Madame Marie-Laure Alves, qui nous a beaucoup aidé. Son écoute et ses conseils nous ont permis de progresser dans notre démarche.
Nous tenons également à remercier vivement notre encadrant, Monsieur Jean Lieber, enseignant chercheur au LORIA, pour son accueil, le temps passé ensemble et le partage de son expertise au quotidien.
Nous remercions également toute l'équipe K et l'équipe Sémagramme pour leur accueil et leur disponibilité, et en particulier Monsieur Emmanuel Nauer, qui nous a beaucoup aidé à comprendre les problématiques de recherche sur lesquelles nous travaillions. Enfin, nous tenons à remercier toutes les personnes qui nous ont conseillé et relu lors de la rédaction de ce rapport : nos familles, nos professeurs et aussi nos camarades de cours.

Sommaire

Table des matières

1 Analyse du problème et organisation **6**
1.1 Le raisonnement à partir de cas . . . 6
1.2 Travail existant . . . 7
1.3 Mise en place du projet . . . 7

2 Conception et développement **8**
2.1 Début du développement . . . 8
2.1.1 Grew . . . 8
2.1.2 WiCorrector . . . 10
2.2 Réflexions sur les fonctionnalités . . . 11
2.2.1 API pour la lecture de fichier . . . 11
2.2.2 Protocole d'ajout de filtres . . . 11
2.2.3 Fichier de sortie . . . 12
2.3 Détail sur les différents filtres . . . 12
2.3.1 Le type GlobalRejector . . . 12
2.3.2 Le type LocalRejector . . . 13
2.3.3 Le type Purifier . . . 13
2.3.4 Travail effectué par LIMSI . . . 14

3 Résultats **15**
3.1 Statistiques . . . 15
3.1.1 Exécution individuelle des filtres . . . 15
3.1.2 Superposition et commutativité des filtres . . . 16
3.2 Résultat final . . . 16
3.3 Ouverture . . . 16

Rappel du sujet

Problématique de recherche :

Le raisonnement à partir de cas (RàPC) est un raisonnement hypothétique (en général) qui consiste à résoudre un nouveau problème (le problème cible, noté cible) en s'appuyant sur une base de cas, un cas étant la représentation d'un épisode de résolution de problème. On appelle cas source un élément de la base de cas. Souvent, on représente un cas source simplement par un couple (srce, sol(srce)) : srce est un problème source, sol(srce) est une solution de ce problème source. Un modèle du processus de RàPC classique comprend deux étapes d'inférence :
Remémoration : un cas source (srce, sol(srce)) jugé similaire au problème cible (par exemple, sur la base d'une distance entre problèmes) est sélectionné.
Adaptation : La solution sol(srce) de ce cas est modifiée en une solution candidate sol(cible) de cible.
La correction de phrases est la problématique de la transformation d'une phrase incorrecte (en particulier, grammaticalement) en une phrase corrigée (nous choisirons la langue française dans ce travail, même si la problématique existe dans toutes les langues). Un cas de correction de phrase est donc un couple (srce, sol(srce)) où srce est une phrase incorrecte et sol(srce) une correction de srce. Par exemple, on a les deux cas :
srce1 = Tu as pas mangé. sol(srce1) = Tu n'as pas mangé.
srce2 = Il a recommencer. sol(srce2) = Il a recommencé.
L'adaptation se fait par des techniques de raisonnement par analogie : la solution sol(cible) est solution d'une équation analogique « srce est à sol(srce) ce que cible est à y ». Par exemple, l'adaptation de (srce2, sol(srce2)) à cible = Tu as manger. consiste à résoudre Il a recommencer. est à Il a recommencé. ce que Tu as manger. est à x qui a pour solution, avec la relation d'analogie utilisée dans le projet, x = Tu as mangé., proposition de solution proposée par le système.

Sujet :
Comme pour tout système à base de connaissances, la qualité d'un système de RàPC dépend de celle de son moteur d'inférences mais également de la qualité de sa base de connaissances, en particulier de sa base de cas. Une bonne base de cas doit avoir plusieurs qualités. Les cas sources doivent être corrects. Elle doit avoir une bonne couverture (et permettre de résoudre correctement une proportion importante de cas). Elle devrait ne pas être trop redondante (certains cas différents correspondent à la même correction).
Pour cela, on pourra consulter les mouchards d'édition de Wikipédia pour en extraire des listes de fautes grammaticales ou orthographiques typiques, ainsi que des sites de dictées ou d'orthographe. Il faudra mettre en place les outils de collecte automatiques, paramétrables en fonction des sites.
Une autre piste est la mise en place d'un jeu interactif avec un but. Le but est de faire corriger des phrases fautives par les joueurs. La phrase corrigée devrait émerger de la majorité des propositions de correction. Les phrases fautives pourraient être extraites de listes d'exemples fautifs, ou produites automatiquement à partir de patrons prédéfinis ou par application de l'analogie sur des cas déjà collectés.
Une courte étude bibliographique sur la maintenance de base de cas permettra de suggérer des pistes pour l'acquisition d'une bonne base de cas. Il faudra mettre en place une méthode pour cela, qui pourra s'appuyer sur les sites mentionnés ci-dessus.

Introduction

Le TAL (traitement automatique des langues) est un domaine d'études dont l'objectif est de créer des outils de traitement de la langue pour diverses applications. Il traite de nombreux sujets notamment dans le domaine de la compréhension automatique des textes, la traduction automatique, ou encore la correction automatique des fautes orthographiques et grammaticales. C'est pourquoi linguistes et informaticiens collaborent étroitement sur l'étude de la langue et le traitement automatique des données. Le traitement automatique du langage couvre donc un vaste champs de disciplines de recherche et se trouve à l'origine de nombreux travaux.

Parmi les premiers travaux, qui remontent aux années 50 pendant la guerre froide, se trouve la mise au point du premier traducteur automatique. Nommé « experience Georgetown-IBM » et développé par l'université de Georgetown en collaboration avec la société IBM, le projet offrait la possibilité de traduire du russe vers l'anglais. C'est en 1954 que la première démonstration sur une soixantaine de phrases s'est faite, une vraie prouesse technologique pour l'époque. Depuis, le traitement automatique des langues a bien évolué et est devenu un domaine pluridisciplinaire. Il peut allier la linguistique, l'informatique, et même se coupler à de l'intelligence artificielle pour créer des applications de plus en plus complexes nous permettant de simplifier nos tâches quotidiennes.

Un correcteur automatique de la langue est un exemple d'outil utilisant des domaines du traitement automatique des langues. Il utilise des disciplines de recherche variées telles que l'analyse syntaxique, la correction orthographique et d'autres plus spécifiques telles que la délimitation de phrase ou la morphologie. Un outil tel que celui-ci est un monstre de conception et de développement, et les meilleurs correcteurs automatiques connus à ce jour ont été développés par des entreprises internationales comme Microsoft, Google ou encore Apple.

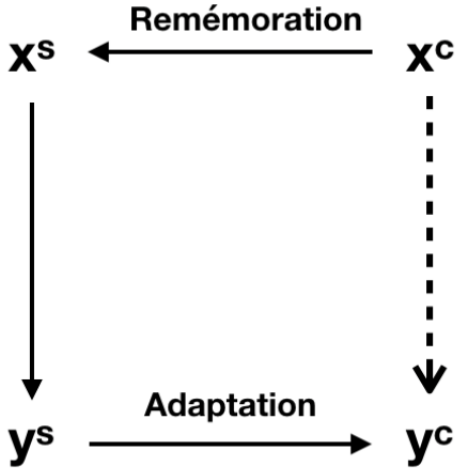
Contrairement aux cas décrits précédemment, notre sujet consiste à créer un système permettant la correction automatique de phrases françaises en s'appuyant sur le raisonnement à partir de cas (RàPC). Pour répondre à cette problématique, nous allons tout d'abord introduire le raisonnement à partir de cas et expliquer comment il va nous permettre de corriger des phrases. Puis nous aborderons une partie plus pratique où nous détaillerons les différentes étapes permettant de résoudre notre problématique. Nous parlerons ensuite des résultats obtenus suite à nos développements et finirons par introduire les suites possibles de notre projet.

1 Analyse du problème et organisation

1.1 Le raisonnement à partir de cas

Le raisonnement à partir de cas (noté RàPC) est un raisonnement qui consiste à résoudre des problèmes en s'appuyant sur des expériences similaires rencontrées par le passé. Grâce aux expériences passées déjà résolues, nous arrivons à en inférer une solution qui s'ajoute à notre expérience. La reproduction de cet exemple un grand nombre de fois nous permettra d'avoir une expérience telle que nous pouvons essayer d'inférer une solution à des problèmes similaires.

Il y a deux principales étapes dans ce raisonnement. Premièrement, il y a ce qu'on appellera la « remémoration », qui est le rapprochement entre notre problème actuel (que nous appellerons problème cible) et un problème qui a déjà été résolu dans le passé (qu'on nommera problème source). La deuxième étape s'appelle « l'adaptation ». Ce principe permet d'obtenir une solution de notre problème source en modifiant la solution du problème cible.



Nous voyons donc qu'il est possible de résoudre des problèmes grâce au raisonnement à partir de cas. Néanmoins, pour pouvoir les résoudre, nous devons avoir des problèmes similaire déjà résolus. Cet ensemble de couples (problème, solution) résolu s'appelle une base de cas. En plus du couple (problème, solution), une explication est ajoutée et permet d'améliorer l'étape de la remémoration.

Voici un exemple de résolution de problème grâce au raisonnement à partir de cas : Ici le problème que nous rencontrons est une phrase contenant une erreur de français (problème cible) : « Tu n'as pas manger. ». Nous avons dans notre base de cas un couple (phrase erronée, phrase corrigée) tel que celui-ci : (« Il a recommencer. », « Il a recommencé. ») où la phrase erronée est le problème source et la phrase corrigée est la solution du problème source. La remémoration va donc rapprocher notre problème cible du problème source et en utilisant la solution du problème source ainsi que l'explication, l'adaptation va pouvoir nous fournir une solution de notre problème cible qui sera « Tu n'as pas mangé. ».

Pour parvenir à résoudre notre problématique qui est la correction automatique du français à l'aide du raisonnement à partir du cas, il nous faut donc une base de cas qui permette en théorie de corriger toutes les erreurs possible du français. En prenant en compte tous les types d'erreurs possibles (grammaire, orthographe, conjugaison, etc.), il y aurait un nombre immense de cas à définir dans notre base, ce qui est impossible à construire à la main. Le but de notre projet est donc de construire une base de cas à l'aide d'outils diverses et variés qui serait totalement automatisée.

Nous voyons donc qu'il est possible de résoudre des problèmes grâce au raisonnement à partir de cas. Cependant, pour pouvoir les résoudre, il est nécessaire d'avoir des problèmes similaires déjà résolus. Cet ensemble de couples (problème, solution) résolus s'appelle une base de cas. En plus du couple (problème, solution), une explication est ajoutée et permet d'améliorer l'étape de la remémoration.

1.2 Travail existant

Une première piste nous a été fournie par un groupe d'étudiants de L3 de 2017-2018 composé de M. Giang, M. Levy et M. Ly, qui avaient travaillé sur un projet intitulé Corrector. Le projet consistait à faire un site WEB capable d'apporter une correction à une phrase erronée donnée en entrée. Cette correction devait se faire à l'aide d'une base de cas qui pouvait s'enrichir avec des interactions humaines (utilisateur/administrateur du site).

Notre début d'étude a donc été guidé par les moyens mis en œuvre pour effectuer un remplissage automatique de leur base de cas initiale, et plus particulièrement un : les corpus de WiCoPaCo [**1**]. Le site WiCoPaCo met en libre accès des fichiers au format XML contenant des phrases, ou parties de phrases avec une correction effectuée ainsi qu'un commentaire éventuel laissé par l'auteur de la correction. Ces fichiers sont le résultat des corrections faites par les administrateurs des pages Wikipédia, ce qui nécessite une correction étant donné que le contenu des pages est apporté par des utilisateurs. Les fichiers en question contiennent donc des centaines de milliers de cas composés de la manière suivante : le groupe de phrase avant modification avec la mise en évidence de la faute, suivi du même groupe de phrase avec la correction apportée également mise en évidence.

L'intérêt principal de ces fichiers étant l'énorme masse de données qu'ils contiennent, nous permettant ainsi d'en extraire un grand nombre de cas. Cependant, même si cette solution semble être idéale et simple à mettre en place, il s'avère qu'elle est loin d'être parfaite. Le problème de ces corrections est qu'une partie se trouve être des corrections de contenu, une autre étant des reformulations, et bien d'autres types de corrections n'étant pas des erreurs de français mais sont pourtant contenues dans ces fichiers. La problématique de l'épuration de cette énorme masse de données se pose donc.

Face à ce problème, le groupe d'étudiants de L3 avait mis en place un script Python qui prenait un fichier XML en entrée et produisait un fichier CSV en sortie. Le script s'occupait aussi de la suppression de certains cas : les retours en arrière. Il ne retenait donc pas les cas qui étaient des retours sur correction, c'est-à-dire lorsque le correcteur transformait une phrase A en phrase B, puis transformait à nouveau la phrase B en phrase A.

C'est donc en reprenant cette base de travail que nous avons débuté notre projet, dans l'optique de pouvoir épurer cette énorme masse de données à l'aide de filtres pour obtenir uniquement des cas de corrections de langue.

1.3 Mise en place du projet

Pour mettre en place notre projet, nous avons donc grandement utilisé le travail déjà effectué par nos collègues qui nous ont précédés. Nous avons décidé d'utiliser l'énorme quantité de cas que nous fournissait les fichiers XML de WiCoPaCo pour créer notre base de cas. Ces fichiers regroupant plus de 400 000 cas, elle serait suffisamment conséquente pour couvrir un grand nombre d'erreurs de français. Néanmoins, comme cela a été expliqué ci-dessus, nous ne pouvons pas seulement transformer ces fichiers directement en une base de cas car un grand nombre de cas n'est pas exploitable. Nous devons donc reprendre le travail qui a été fait en amont et continuer à filtrer les cas jusqu'à obtenir un ensemble de cas corrects.

À la suite d'une réflexion sur le développement de notre projet, nous avons décidé de ne pas reprendre les travaux effectués par les étudiants précédent pour plusieurs raisons : premièrement, bien que nous comprenions l'idée directrice du développement, nous n'avions pas toutes les subtilités pour comprendre parfaitement le code. De plus, nous avions dans l'optique d'implémenter plusieurs filtres afin de rendre la création de ceux-ci plus facile. Nous nous sommes donc résolus à utiliser le langage orienté objet Java pour le développement de notre application. C'est un langage que nous avons eu l'habitude d'utiliser au cours de nos études. Il permet de lire et d'écrire des fichiers aisément et permet, une fois le projet structuré, une implémentation simple et rapide de nouvelles fonctionnalités.

Cependant, notre projet consistant à trouver la correction d'une phrase de français en utilisant une base de cas, nous n'avons par conséquent pas accès à un analyseur syntaxique ou orthographique pour pouvoir détecter si une phrase contient des erreurs. Nos encadrants nous ont demandé de réfléchir à une méthode permettant d'effectuer cette détection.

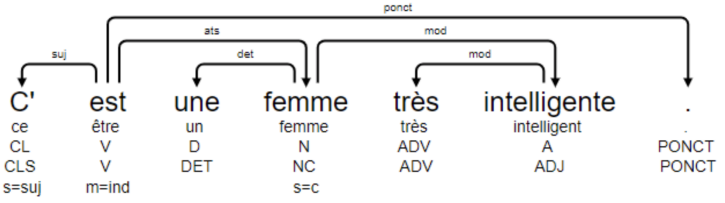
2 Conception et développement

2.1 Début du développement

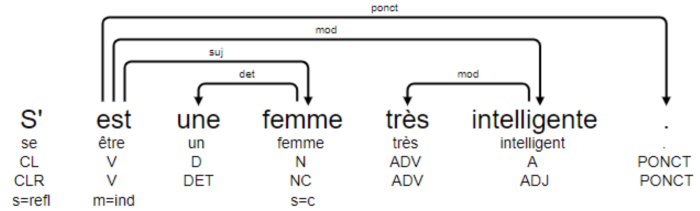
Suite à la mise en place de notre sujet, nous nous sommes donc concentrés sur deux axes principaux : la découverte d'un outil permettant la détection d'erreurs dans une phrase et le début du développement de notre projet en Java.

2.1.1 Grew

Le premier de nos deux axes était la découverte d'un outil permettant l'analyse des phrases pour en trouver les potentielles fautes de français. Pour cela, nos encadrants nous ont orientés vers GREW [**2**], un logiciel dont un de nos encadrants, Bruno Guillaume, connaît bien le fonctionnement. Ce logiciel permet, à partir d'une ou plusieurs phrases données en paramètres, de construire un graphe à partir des lemmes de chaque mot ainsi que grâce au contexte des phrases. Ce graphe permet de déduire les liens que chaque mot entretient avec les autres mots de la phrase. Voici un exemple de graphe que crée l'outil GREW à partir de la phrase « C'est une femme très intelligente. »



Nous pouvons donc retrouver dans ce graphe le lemme de chaque mot, situé sous le mot correspondant ainsi que les liens qu'il a avec les autres, représentés par des flèches qui lient les mots entre eux. Cet outil n'est donc pas un outil de détection d'erreur à proprement parlé. Cependant il permet, si une phrase n'est pas bien assemblé ou dans les cas où il n'arrive pas à trouver les liens entre les mots d'une phrase, de créer des graphes où tous les mots ne sont pas liés entre eux. Prenons par exemple la phrase utilisée dans le graphe précédent et remplaçons le mot « C'est » par « S'est », ce qui est une erreur classique de français. Dans ce cas, GREW réalise le graphe suivant :



Nous voyons ici que GREW n'a pas réussi à lier le 'S' avec les autres mots. Nous dirons par la suite qu'un graphe dans lequel un ou plusieurs mots ne sont liés à aucun autre est un graphe qui représente une qui comporte une ou plusieurs erreurs de français. Notre objectif est donc d'observer les résultats de GREW sur des exemples multiples et variés, et d'analyser ses performances.

Tout d'abord, nous avons dû installer GREW, ce qui fut complexe aux vu des dépendances de ce dernier. L'installation nous a permis d'utiliser l'outil en ligne de commande, puis par l'intermédiaire d'un script BASH, qui nous a permis d'automatiser son utilisation. L'objectif de ce script était de faire analyser un grand nombre de phrases par GREW, correctes ou incorrectes, et de vérifier par le biais de fichier de sortie, le pourcentage de phrases bien analysées par celui-ci. Pour se faire, nous avons regroupé les cas du fichier XML de WiCoPaCo dans un fichier CSV. Ces cas étaient représentés comme sur la capture d'écran ci-dessous :

```
 A la Sorbonne , Meillet surveille le travail Milman Parry .
 À la Sorbonne , Meillet surveille le travail de Milman Parry .
ortho
```

Chaque cas est composé de 3 éléments : une phrase fausse, une phrase corrigée, ainsi qu'un commentaire expliquant la nature de la correction. Sur cette image, la première ligne correspond à la phrase avec erreur, la deuxième correspond à la phrase corrigée et la dernière au commentaire.

De plus, une étudiante de l'école Télécom Saint-étienne, Mme Isabelle Mornard, est venue effectuer un stage au sein du Loria, sous l'encadrement de M. Guillaume, M. Lepage, M. Lieber et M. Nauer. Son objectif en tant que stagiaire était de nous aider à comprendre les différentes erreurs possibles de la langue française ainsi que de les catégoriser. En plus de cela, un de ses objectifs était de créer à la main une base de cas d'environ 200 cas regroupant des erreurs de chaque catégories. La base de cas a été remplie dans un fichier Python et chaque cas était représenté comme ci-dessous :

```
c1.x = "Il a trop de bruit pour travailler."
c1.y = "Il y a trop de bruit pour travailler."
c1.expl = syntaxe("","y ")
cl.prov = https://www.umoncton.ca/umcm-caf/files/umcm-caf/wf/erreurs_a_eviter_no_1.pdf
```

Un cas dans ce fichier est représenté par 3 ou 4 lignes : la première est une phrase avec une erreur, la deuxième est une phrase correcte, la troisième est l'explication de la correction et la dernière ligne est la provenance de l'exemple. La provenance est souvent un site web de correction de français. Nous avons donc utilisé notre script sur la base de cas d'Isabelle ainsi que sur notre base de cas pour tester l'outil GREW. Les fichiers résultants de ces tests nous ont permis de mieux comprendre son fonctionnement ainsi que ses limites.

	Base de cas d'Isabelle		Base de cas de WiCoPaCo	
	Cas ne comportant des erreurs	Cas ne comportant pas d'erreurs	Cas comportant des erreurs	Cas ne comportant pas d'erreurs
Nombre de cas rejetés / nombre de cas	101 / 202	53 / 102	279 / 500	273 / 500
Pourcentage de cas rejetés	50 %	26 %	56%	56%

Les résultats que nous analysons dans les fichiers de sorties nous montrent que dans les cas contenant une erreur, le logiciel GREW détecte cette erreur une fois sur deux. Cependant, ces résultats nous

montrent aussi que le logiciel détecte une erreur 40 % du temps sur des cas ne comportant pas d'erreur. Néanmoins, nous avons réussi à trouver des schémas de phrase qui se répètent et pour lesquelles le logiciel ne marche pas correctement. Par exemple, tous les cas comportant des caractères spéciaux tel que des accolades, des parenthèses, des tirets, des point-virgule, des points d'interrogation, etc. sont des cas pour lesquels GREW n'arrive pas à créer un graphe complet correspondant et donc est considéré comme un cas contenant une faute même si ce n'est pas toujours le cas. A contrario, les cas contenant des erreurs de conjugaisons sont des cas où un graphe complet est parfois créé et donc considéré comme sans faute. Nous pouvons donc en conclure que le logiciel GREW n'est pas infaillible quant à la détection d'erreur mais que des améliorations peuvent être effectuées sur les cas pour pouvoir obtenir de meilleurs résultats. Il est important de noter que GREW n'est pas un outil développé pour détecter les erreurs de français.

2.1.2 WiCorrector

Notre second axe était la mise au point d'un outil simple permettant de traiter les données des fichiers WiCoPaCo. Ces derniers se présentant sous la forme suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<modifs language="fr" wp_dump_date="2007-01-01" wp_first_page_id="1" wp_last_page_id="3000000">
      <modif id="1" wp_page_id="3" wp_before_rev_id="1350936" wp_after_rev_id="1409789" wp_user_id="3763"
wp_user_num_modif="15342" wp_comment="correction de numéraux">
        <before>Antoine Meillet ( né Paul-Jules-Antoine Meillet 11 novembre 1866-21 septembre 1936 ) est le principal
linguiste français des premières décennies du <m num_words="1">XXe</m> siècle .</before>
        <after>Antoine Meillet ( né Paul-Jules-Antoine Meillet 11 novembre 1866-21 septembre 1936 ) est le principal
linguiste français des premières décennies du <m num_words="2">XX e</m> siècle .</after>
  </modif>
        ... Répétition de balises <modif></modif> ...
</modifs>
```

Les balises :

```
<?xml version="1.0" encoding="UTF-8"?>
```

ainsi que

```
<modifs language="fr" wp_dump_date="2007-01-01" wp_first_page_id="1" wp_last_page_id="3000000">
```

donnent des informations générales sur le document qui ne sont pas essentielles pour le traitement que nous désirons effectuer. Puis, suit la balise :

```
        <modif id="1" wp_page_id="3" wp_before_rev_id="1350936" wp_after_rev_id="1409789" wp_user_id="3763"
wp_user_num_modif="15342" wp_comment="correction de numéraux">
```

qui contient le contenu qui nous intéresse puisque c'est elle qui contient le cas. le document se construit de la manière suivante :

```
        <modif ...>
            <before> Contexte autour du morceau de phrase qui s'apprête à être modifié <m
num_words="X"> morceau de phrase qui va être modifié </m> suite du contexte </before>
            <after> Contexte autour du morceau de phrase qui a été modifié <m
num_words="X"> morceau de phrase modifié </m> suite du contexte </after>
        </modif>
```

Donc chacune de ces balises pouvait potentiellement correspondre à un cas qui aurait pu s'ajouter à notre base de cas. Or, après une étude de ces fichiers nous avons observé que chaque balise <modif> n'était pas forcément une correction de langue, c'est pourquoi il était nécessaire d'effectuer un traitement sur les données.

Donc nous avons entrepris la conception d'un logiciel qui porte le nom de WiCorrector. Celui-ci prend en entrée un fichier au format XML tiré du site WiCoPaCo, en extrait les cas puis les traitent de sorte à enlever ceux qui ne constituent pas un cas intéressant pour la création d'une base de cas. Enfin, il retourne les cas retenus dans un fichier au format facilement exploitable.

Ce logiciel allait contenir plusieurs filtres pouvant être répartis dans des catégories bien précises, donc nous nous sommes tournés vers un langage orienté objet possédant des propriétés telles que l'héritage. Le langage Java semblait donc bien correspondre à nos besoins grâce à son polymorphisme de variable, de plus il est possible de trouver en ligne de nombreuses API pour l'exploitation de fichier XML.

Après réflexion, nous avons donc décidé de mettre en place 3 catégories distinctes de filtres :

— Le « globalRejector » qui accepte ou rejette un cas en fonction de l'ensemble du fichier, c'est-à-dire en fonction des autres cas de ce même fichier.

— Le « localRejector » qui accepte ou rejette un cas en ne regardant que le cas lui même.

— Le « Purifier » qui va réduire la taille du cas en supprimant le surplus de caractères non essentiels.

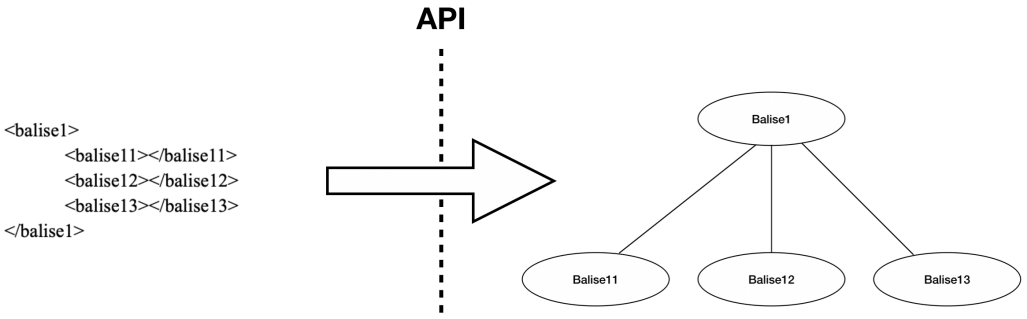
Ces 3 types de filtres s'appliquent de manière consécutive par catégorie sur chaque cas extrait des fichiers WiCoPaCo. Une fois l'épuration effectuée, le logiciel crée un fichier de sortie contenant les cas n'ayant pas été rejetés. Concernant les données de sorties du logiciel, nous avons décidé d'utiliser le format CSV afin de pouvoir facilement insérer son contenu dans une base de données par la suite si besoin.

2.2 Réflexions sur les fonctionnalités

Dans cette partie nous aborderons nos principales réflexions qui ont orienté le développement du logiciel en Java.

2.2.1 API pour la lecture de fichier

Comme les fichiers de WiCoPaCo sont au format XML, nous avons cherché des API permettant une exploitation aisée et pratique du contenu qu'ils renferment. Nous avons donc utilisé une API qui permet de représenter en mémoire le fichier XML en un objet en structure d'arbre. Chaque noeud de cet arbre est une balise, et les enfants de ce noeud correspondent aux balises que ce noeud contient (cf illustration suivante).



Cette API nous permet donc une facilité de développement pour le parcours et l'exploitation des balises ainsi que leurs contenus.

2.2.2 Protocole d'ajout de filtres

C'est ici que nous allons traiter le cœur du sujet : la manière dont les filtres ont été conçus. Ces filtres étant un point crucial dans l'obtention d'un résultat intéressant pour notre recherche, il semblait

plus que nécessaire de mettre en place un protocole rigoureux pour la validation des filtres. Le protocole était assez simple, mais n'en demeurait pas moins efficace.

Pour commencer, nous examinions le contenu du fichier WiCoPaCo dans le but de trouver plusieurs cas qui ne correspondaient pas à nos critères de sélection. Une fois le cas relevé, il fallait l'analyser et comprendre pourquoi il ne concordait pas avec nos attentes, de sorte à trouver une règle nous permettant de détecter ce type de cas. Après quoi, nous nous penchions sur la manière dont le filtre allait être implémenté en s'appuyant sur la règle que nous avions élaborée.

Ensuite, une fois le filtre développé, une multitude de tests devaient être effectués afin de s'assurer que celui ci était bien fonctionnel. Nous exécutions donc le filtre sur un extrait du fichier de WiCoPaCo contenant quelques centaines de cas, et nous observions ceux rejetés. Si le filtre se comportait comme nous l'attendions sur cette base test de taille réduite, nous considérions que le filtre était fonctionnel. Si le résultat se révélait être imparfait, nous entreprenions un ajustement de la règle jusqu'à obtention du résultat escompté.

Puis, une fois le filtre opérationnel, une analyse statistique de l'exécution de ce filtre, appliqué avec les autres filtres déjà existant, sur le fichier entier était effectuée. Le but de la démarche était de démontrer l'utilité de ce dernier malgré l'existence d'autres filtres. Si les résultats statistiques attestaient de l'intérêt du filtre, alors ce dernier était accepté et ajouté au projet.

Enfin, nous appliquions tous les filtres sur les quelques 400 000 cas et entreprenions une nouvelle analyse sur le fichier de sortie du logiciel en quête de cas ne correspondant pas à nos critères pour recommencer le protocole. Le fonctionnement de chacun de ces filtres sera détaillé plus bas (cf. partie 2.3).

2.2.3 Fichier de sortie

Le logiciel, une fois les filtres appliqués sur un fichier passé en entrée, devait produire une sortie exploitable facilement. C'est pourquoi nous avons décidé d'utiliser le format CSV. C'est un format simple, exportable, lisible par un humain sous forme de tableau, et qui peut être aisément inséré dans une base de données via un script si besoin en est. Chaque ligne correspond à un cas, et le fichier est composé de trois colonnes : la première correspond au cas avant que la correction soit effectuée, la deuxième étant le cas avec la correction, et la troisième colonne se trouve être l'éventuel commentaire que l'auteur de la correction peut faire sur la correction qu'il effectue sur Wikipedia. Il est également possible d'activer la sortie pour chacun des filtres, chaque fichier de sortie porte le nom de « RejectedBy » concaténé avec le nom du filtre et contient les cas rejetés. Cela permet de voir ce que chaque filtre a effectué lors de l'exécution.

2.3 Détail sur les différents filtres

2.3.1 Le type GlobalRejector

Comme expliqué de manière succincte plus haut, le type de filtre GlobalRejector est une catégorie contenant les filtres qui nécessite une analyse entière du document d'entrée, pour y collecter des informations, afin de statuer sur l'acceptation d'un cas. Dans cette catégorie se trouve deux filtres : Le premier étant le « RollbackFilter », qui a pour but de refuser les retours sur correction. Le filtre parcourt une première fois entièrement le fichier en répertoriant chaque correction, puis lors du parcours du document pour le traitement des cas, le filtre scrute ses données pour voir si une modification inverse est effectuée à un moment. S'il existe une modification inverse dans le document, alors le cas est rejeté. Voici un exemple que le RollbackFilter rejette : On trouve une première modification comme suit,

```
<before>Elle a <m num_words="1">agi</m> pendant de nombreuses années par la voie armée...</before>
<after>Elle a <m num_words="1">agit</m> pendant de nombreuses années par la voie armée...</after>
```

Puis, plus loin dans le document on trouve la correction inverse,

```
<before>Elle a <m num_words="1">agit</m> pendant de nombreuses années par la voie armée...</before>
<after>Elle a <m num_words="1">agi</m> pendant de nombreuses années par la voie armée...</after>
```

Comme le fichier contient la correction « agi » en « agit », puis « agit » en « agi » sur le même cas, le filtre rejette les deux corrections puisqu'il ne peut pas déterminer laquelle de ces deux corrections est juste.

2.3.2 Le type LocalRejector

Quant au type « LocalRejector », c'est une catégorie comprenant des filtres qui inspecte uniquement le cas et son contenu pour décider si le cas est rejeté. Deux filtres ont été implémentés dans cette catégorie :
Le premier porte le nom de « EstheticalRestructurationRejector » et a pour but de détecter les reformulations ou restructuration de phrases. Pour ce faire, notre filtre utilise la distance d'édition de Levenshtein, qui au sens mathématique du terme, est une distance donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. C'est pourquoi si cette distance est importante par rapport à la taille des mots modifiés, nous pouvons considérer que le mot entier a été changé et que ce n'était pas une simple correction orthographique.
Notre filtre a été implémenté de la manière suivante, un cas est rejeté si :

— La longueur de la plus petite chaîne de caractères entre l'avant et l'après correction est inférieure à 4 caractères, et que la distance de Levenshtein est plus grande que la moitié de la plus petite chaîne.

Ou, si

— La longueur de la plus petite chaîne de caractères entre l'avant et l'après correction est supérieure ou égale à 4 caractères, et que la distance calculée est plus grande qu'un tiers de la plus petite chaîne.

Le choix d'une telle implémentation résulte d'une première réflexion sur les possibles fautes qu'il est possible de faire, et plus précisement sur le rapport entre la taille de la faute et la taille du mot en question. Puis cette réflexion a été testée de manière empirique sur 500 cas d'une base test, ou chaque cas rejeté ou accepté ont été analysé pour ajuster les paramètres du filtre (seulement quelques fautes de conjugaison sont filtrés à tort). Voici un exemple que le EstheticalRestructurationRejector rejette :

```
<before>Il <m num_words="1">tient</m> son nom de la rivière éponyme .</before>
<after>Il <m num_words="1">tire</m> son nom de la rivière éponyme .</after>
```

Le deuxième, « NumberRejector » élimine les corrections portant sur les chiffres, car ce sont des corrections de contenu (date, pourcentage, ...) et donc, elles ne constituent pas d'erreurs de français. Voici un exemple que le NumberRejector rejette :

```
<before>...ceux-ci devant renoncer à <m num_words="1">75</m> % de leurs créances .</before>
<after>...ceux-ci devant renoncer à <m num_words="1">78</m> % de leurs créances .</after>
```

2.3.3 Le type Purifier

En plus de ces deux filtres, nous avons ajouté une autre catégorie, « PurifierFilter », qui n'est pas un filtre en soit car il rejette aucun cas. Ce « purificateur » a deux principaux objectifs : premièrement, il sert à « épuré » les cas qui seront retenu par les filtres précédent. Cette épuration permettra d'alléger

les données présent dans la base de cas mais surtout d'améliorer les étapes de remémoration ainsi que d'adaptation en enlevant des caractères inutiles. De plus, nous avons vu précédemment dans l'utilisation du logiciel GREW que celui-ci n'arrivait pas à construire de graphe complet lorsque la cas étudié comportait des caractères spéciaux. Cette épuration permettra aussi d'améliorer la qualité d'analyse de GREW pour la détection d'erreurs.

Dans cette catégorie, nous retrouvons deux filtres. Le premier filtre, « SentencePurifier » a pour mission de supprimer tout le contenu superflu autour de la correction, et de n'en garder que le noyau. Le problème étant que les cas contenus dans le fichier de WiCoPaCo sont souvent entourés du contexte, qui n'est pas important pour l'utilisation que nous voulons en faire. C'est pourquoi nous avons décidé de supprimer le contexte, dans l'optique d'alléger le fichier de sortie Par exemple le cas :

```
<before>L' histoire de l' algèbre linéaire moderne commence dans les années 1843 et 1844 . En 1843 , William
Rowan Hamilton ( inventeur du terme vector ) découvre les quaternions . En 1844 , Hermann Grassmann publie
un livre Die lineare <m num_words="1">Ausdehnungsleshre</m> .</before>
<after>L' histoire de l' algèbre linéaire moderne commence dans les années 1843 et 1844 . En 1843 , William
Rowan Hamilton ( inventeur du terme vector ) découvre les quaternions . En 1844 , Hermann Grassmann publie
un livre Die lineare <m num_words="1">Ausdehnungslehre</m> .</after>
```

va se transformer par l'application du filtre SentencePurifier en :

```
<before>En 1844 , Hermann Grassmann publie un livre Die lineare <m num_words="1"> Ausdehnungsleshre
</m> .</before>
<after>En 1844 , Hermann Grassmann publie un livre Die lineare <m num_words="1">Ausdehnungslehre
</m> .</after>
```

Enfin, le filtre SpecialCharacterPurifier, est assez simple. Il consiste à supprimer les caractères spéciaux suivant : « $ », « * », « / », « # », se trouvant au début de la première phrase des cas. Les corrections pouvant être effectuées sur tout le contenu de la page, certains cas sont des annotations en pied de page et ont donc une mise en forme particulière. Ce filtre n'a d'intérêt que s'il est appliqué après le filtre « SentencePurifier », car il risquerait d'effectuer le même travail

2.3.4 Travail effectué par LIMSI

Grâce à nos recherches effectuées sur WiCoPaCo, nous avons pu trouver des travaux effectués par des chercheurs de LIMSI[**3**]. Ces travaux [1] leurs ont permis d'extraire automatiquement des fichiers XML de WiCoPaCo un corpus d'erreurs composé de 72 483 erreurs lexicales et 74 100 erreurs grammaticales. Leurs résultats peuvent être extraits à partir d'un fichier XML regroupant les 150 000 cas, présenté sous la forme :

```
<annotation>
  <modif_id>161673</modif_id>
  <label>real_word_error</label>
</annotation>
```

Chaque cas est représenté par un identifiant « id », correspondant à l'identifiant du cas dans le fichier XML de WiCoPaCo. De plus, une étiquette est associée afin de préciser si ce cas est une erreur lexicale (non-word errors) ou grammaticale (real-word errors). Nous avons donc associé leurs travaux aux nôtres en créant un filtre s'intitulant « SpellingErrorLabel », bien qu'il ne soit pas à proprement dit un filtre car il peut uniquement s'appliquer sur la base de cas de WiCoPaCo. Ce filtre permet de ne garder que les cas dont l'identifiant est inscrit dans le fichier résultant des travaux de LIMSI.

Cependant, leur fichier représente quelques problèmes : premièrement, leur fichier de résultats est un travail obtenu à partir de l'analyse du fichier de WiCoPaCo. Ce fichier sera donc utile seulement

1. **https://wicopaco.limsi.fr/pub/taln10.pdf**.

si nous analysons les cas de WICoPaCo, mais sera inutile si nous prenons une autre base de cas à analyser. Deuxièmement, leur travail est muni d'un document expliquant leurs développement mais celui-ci n'est pas très explicite quant à la nature de leurs filtres. Nous ne pouvons donc pas assurer que leurs travail ne supprime pas de cas qui nous serait utiles. Nous nous questionnons donc sur l'utilité de ce filtre pour notre projet.

3 Résultats

Dans cette dernière partie, nous allons donc vous montrez les résultats que nous avons obtenus suite au développement de WiCorrector. Nous aborderons aussi les problèmes que nous avons rencontrés au cours de cette initiation à la recherche ainsi que les possibles suites qui peuvent être données à nos travaux.

3.1 Statistiques

3.1.1 Exécution individuelle des filtres

Dans cette partie nous donnerons les résultats de chacun de nos filtres en exécution seul sur le fichier de WiCoPaCo complet.

Catégorie de filtres rejetant des cas :

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre RollbackFilter	154 449 / 408 816	37,7%
Filtre EstheticalRestructurationRejector	145 141 / 408 816	35,5%
Filtre NumberRejector	47 194 / 408 816	11,5%

Notez que l'exécution de chaque filtre ne prend pas plus de quelques minutes à se faire sur tout le fichier (en dehors du RollbackFiltrer qui prend plusieurs dixaines de minutes à cause sa complexité algorithmique).

Catégorie de filtres supprimant le contenu superflu d'un cas :

	Nombre de caractèr supprimés / nombre de caractères traités	Pourcentage de caractères supprimés
Filtre SentencePurifier	28 740 736 / 119 554 961	24%

	Nombre de caractèr supprimés / nombre de phrases traitées	Pourcentage de caractères supprimés par phrase
Filtre SpecialCaracterPurifier	191 163 / 817 632	23,3%

Quant aux filtres « Purifier », leurs exécutions ne dépasse pas quelques dixaines de secondes.

3.1.2 Superposition et commutativité des filtres

Il est intéressant de voir si l'ordre d'application des filtres a une incidence sur le fichier de sortie. Cela permet de s'assurer que les filtres, entre eux, ne s'entravent pas dans l'exécution de leurs fonctions. Nous allons donc exécuter les filtres rejetant des cas dans des ordres différents sur le fichier complet.

Ci-dessous se trouve le tableau des résultats de l'exécution la catégorie de filtres « GlobalRejector », puis la catégorie de filtres « LocalRejector » sur le fichier WiCoPaCo complet. Les filtres ont étés appliqués dans le même ordre que celui d'apparition dans le tableau.

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre RollbackFilter	154 449 / 408 816	37,7%
Filtre NumberRejector	27 269 / 254 366	10,7%
Filtre EstheticalRestructurationRejector	99 319 / 227 097	43,7%

Nous obtenons donc un fichier de sortie comportant 127 779 cas.
NB : Dans cette configuration, l'exéuction complète a durée une quarantaine de minutes sur une de nos machines.

Ci-dessous se trouve le tableau des résultats de l'exécution la catégorie de filtres « LocalRejector », puis la catégorie de filtres « GlobalRejector » sur le fichier WiCoPaCo complet. Les filtres ont étés appliqués dans le même ordre que celui d'apparition dans le tableau

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre NumberRejector	47 194 / 408 816	11,5%
Filtre EstheticalRestructurationRejector	148 485 / 361 622	41%
Filtre RollbackFilter	85 358 / 213 138	40%

Nous obtenons donc un fichier de sortie comportant 127 779 cas.
NB : Dans cette configuration, l'exéuction complète a durée une douzaine de minutes sur une de nos machines.

Après comparaison des deux fichiers de sorties, nous avons constatés qu'ils étaient identiques en nombre de cas, mais aussi en contenu. Nous pouvons donc supposer que les filtres sont commutatifs. Cela se démontre assez facilement puisque les filtres que nous avons implémentés ne traitent que le contenu de la balise encadrant la partie corrigée, et ce de la mÃªme manière que ce soit avant ou après la correction. Donc on peut dire que l'ordre d'exécution des filtres a peu d'importance, si ce n'est le temps d'exécution du logiciel (environ trois fois plus rapide dans une configuration que dans l'autre).

3.2 Résultat final

contenu : comparaison entre le fichiers WiCoPaCo de base et la pseudo base de cas obtenue :
— Nombre de cas enlevé au total
— Nombre de cas épuré au total
— Temps de traitement des données

3.3 Ouverture

contenu :
— Les difficultés rencontrés au cours de notre initiation à la recherche
— Le travail qu'il reste à accomplir (finir l'épuration de la base de cas puis remémoration et adaptation pour obtenir un logiciel fonctionnel)

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre SpellingErrorLabelFilter	269 943 / 408 816	66%

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre SpellingErrorLabelFilter	269 943 / 408 816	66%
Filtre RollbackFilter	46 623 / 138 873	33,5%
Filtre NumberRejector	0 / 92 250	0%
Filtre EstheticalRestructurationRejector	2 113 / 92 250	2,3%

	Nombre de cas rejetés / nombre de cas traités	Pourcentage de cas rejetés
Filtre NumberRejector	47 194 / 408 816	11,5%
Filtre EstheticalRestructurationRejector	148 485 / 361 621	41%
Filtre RollbackFilter	85 358 / 213 137	40%
Filtre SpellingErrorLabelFilter	57 551 / 127 779	45%

Références

[1] Wikipedia Correction and Paraphrase Corpus `https://wicopaco.limsi.fr`

[2] Graph Rewriting for Natural Language Processing `http://grew.fr`

[3] Laboratoire d'informatique pour la mécanique et les sciences de l'ingénieur `https://www.limsi.fr/fr/`