Outils d’aide à l’écriture numérique : détection d’erreur et correction orthographique

L1 Informatique – Électronique
Année 2022/2023 – Groupe 1

Eliaz ANDRE LE POGAMP
Pierre GINGUENET

Au quotidien, si l’on a besoin de mettre en forme des textes, prendre des notes de cours, rédiger un rapport, ou juste plus généralement d’écrire, il est commun de lancer un logiciel que l’on appelle « Éditeur de texte ».
Il en existe une grande quantité, ils sont très variés et présents sous tous les systèmes d’exploitation. LibreOffice pour Windows, Brackets sur Mac et Gedit sur Ubuntu, le système d’exploitation Linux, en sont certainement les plus connus et utilisés.

Les utilisateurs le savent, lorsqu’ils écrivent, ils finissent immanquablement par faire des erreurs. Celles-ci peuvent être des fautes de frappe, qui sont un appui sur une mauvaise touche du clavier, ou plus simplement des fautes d’orthographe, comme le pourrait être le fait d’écrire « commète » à la place de « comète ». Dans ces deux cas, l’éditeur de texte fera remarquer la faute à l’utilisateur. Le plus souvent, il mettra en évidence le mot mal écrit pour y attirer l’attention de l’utilisateur et l’inciter à vérifier la bonne écriture du mot.

Il existe un troisième cas dans lequel l’éditeur de texte voudra mettre en évidence une erreur, l’utilisation de noms propres. Ainsi, bien que certains noms propres qui sont aussi des noms communs puissent passer au travers de la mise en évidence, comme « Renard » ou « Olivier », la plupart d’entre eux ne le fera pas et ce en dépit de leur écriture tout à fait correcte. Le cofondateur de Microsoft, Bill Gates, verra ainsi son nom souligné en rouge dans LibreOffice bien qu’il soit écrit sans faute, comme illustré sur la figure ci-contre.

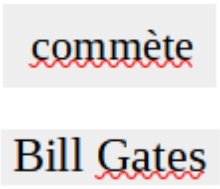

Figure 1: Mise en évidence d'une faute sur l'éditeur de texte LibreOffice

Cette mise en évidence d’erreur qui se traduit dans notre exemple par un soulignage rouge s’appelle la détection d’erreur, « spell checking » en anglais. Elle va de pair avec une autre fonctionnalité de l’éditeur de texte, la correction d’erreur, qui peut être passive ou active.

Dans ce document, nous aborderons dans la première section la détection d’erreur. Dans un second temps, il sera question de la correction d’erreur. Nous en déterminerons les composants et le fonctionnement, l’utilité étant évidente. En général, personne ne souhaite laisser des fautes d’orthographe dans le texte qu’il écrit.

Cette première section va traiter de la détection d'erreur. Comme évoquée dans l'introduction du document, cette fonctionnalité n'a aucun impact sur le texte en lui-même, ce qui implique qu'aucun caractère ne doit être modifié lorsqu'il s'agit de la détection d'erreur.
Le procédé de mise en évidence d'une erreur est divisible en 3 étapes, que nous allons à présent détailler.
Tout d'abord, pour que la détection d'erreur puisse s'exercer, il faut en quelque sorte la « nourrir » avec du texte. Que l'on écrive petit à petit dans l'éditeur comme on écrirait une phrase, ou bien que l'on y colle un bloc de texte déjà écrit, la procédure sera similaire. L'éditeur devra dans un premier temps séparer les phrases en mots pour pouvoir déterminer au cas par cas s'ils nécessitent une mise en évidence ou non. Cette opération s'appelle la **segmentation** en français, la « **tokenization** » en anglais.[1][2] Elle est réalisée par un « tokenizer » et les mots obtenus sont appelés des « tokens ».
La segmentation est un processus simple, réalisée à l'aide de fonctions parmi lesquelles la fonction Python split()[3]. Associée à un String, cette fonction renvoie la liste de String qui composaient le String originel, mais qui ont été séparés.
L'élément séparateur peut être passé en paramètre de la fonction. S'il est laissé vide, alors ce sera l'espace «␣». Si la lettre « i » est passée en paramètre, alors « i » sera l'élément séparateur.

```
txt = "Je suis un Tokenizer!"
x = txt.split()
print(x)
```

```
['Je', 'suis', 'un', 'Tokenizer!']
```

```
txt = "Je suis un Tokenizer!"
x = txt.split("i")
print(x)
```

```
['Je su', 's un Token', 'zer!']
```

Figure 2: Fonctionnement du tokenizer [3]

Passé cette étape, le texte n'est à présent plus qu'une liste de mots, des « tokens », aussi longue qu'il y avait de mots dans le texte. Il est désormais possible de vérifier si chaque mot est connu de l'éditeur ou non.

Il y a plusieurs façons de le faire. En réalité, les variations d'une langue à une autre sont telles qu'une solution adaptée pour le français pourrait ne pas l'être du tout pour une langue plus exotique. Il faut donc concilier fiabilité de la détection et rapidité de l'opération, pour que l'éditeur ne passe pas plusieurs secondes à vérifier chaque mot, ni ne commence à considérer comme faux des mots correctement écrits, car les deux seraient très incommodants.
De ce fait, peu importe la langue, la détection d'erreur a un socle commun dans l'éditeur de texte : le dictionnaire. Il est semblable à son homonyme papier, dans le sens où il prend la forme d'une longue liste de mots triés dans l'ordre alphabétique. Néanmoins, il ne mène à aucune définition, ce qui est généralement l'usage qu'on se fait d'un dictionnaire. Plus il est long, plus il a d'éléments et donc plus il est complet.

Il faut savoir que dans notre cas, utiliser le terme « dictionnaire » comme étant un élément unique est un abus de langage : en réalité il n'existe pas un mais plusieurs dictionnaires, qui permettent chacun de couvrir des mots différents.
Le premier dictionnaire est qualifiable d'immuable. Il est généralement le plus grand. Il est présent dans le fichier alloué à l'éditeur de texte sur l'ordinateur. Par exemple, sur Windows, le dictionnaire français de LibreOffice est situé dans un fichier nommé «dict-fr» auquel on peut accéder en suivant le chemin Windows/Programmes/LibreOffice/share/extensions/dict-fr.[4][5] Ce dictionnaire

contient tous les mots que l'on peut trouver dans un dictionnaire classique. Noms communs, adjectifs, verbes y sont inscrits.

Le second dictionnaire est légèrement différent car il est lui totalement modélisable selon la volonté de l'utilisateur : il est personnalisable. Ce dictionnaire annexe permet de passer outre un gros problème de la correction orthographique mentionné dans notre introduction. À savoir, comment est-il possible de vérifier la bonne écriture des noms propres tels que des prénoms ou des noms d'entreprises étant donné qu'on ne les connaît pas tous et que certains ne sont même pas encore inventés ? La solution est apportée par ce dictionnaire : grâce à la fonctionnalité d'ajout d'un mot au dictionnaire par l'utilisateur, il est possible de faire en sorte que l'éditeur, lors de l'analyse de ce mot – qui n'est pas dans son dictionnaire «principal» – ne le détecte pas comme une erreur. En effet, l'éditeur va ajouter le mot sélectionné par l'utilisateur à son dictionnaire annexe et ne le considérera à l'avenir plus comme une erreur.

La figure ci-contre illustre la fonctionnalité d'ajout au dictionnaire que l'on peut rencontrer sur LibreOffice lorsque l'on fait un clic-droit sur un mot signalé comme une erreur. Ladite fonctionnalité est visible en bleu.

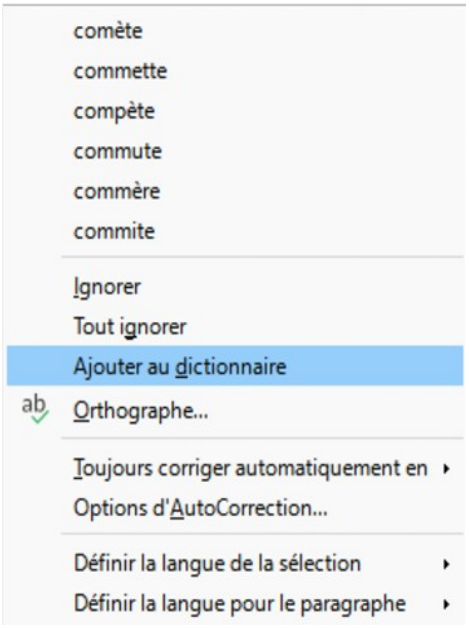

Figure 3: Fonctionnalité d'ajout au second dictionnaire

Ce dictionnaire personnalisable peut par ailleurs lui aussi être consulté par l'utilisateur s'il est curieux de savoir quels mots il a pu ajouter à son dictionnaire personnel au cours de l'utilisation. Il est cependant situé à un endroit différent, dans un dossier caché, l'AppData. Sur Windows, on peut y accéder en effectuant le raccourci Windows + R et en écrivant «%appdata%» dans la barre qui s'ouvre. Dans l'explorateur de fichiers qui s'ouvre, repérer le dossier LibreOffice et suivre le chemin LibreOffice/4/user/wordbook/.[4][5] On a alors à disposition un fichier sous format DIC (pour dictionary) que l'on peut ouvrir pour consulter ses propres mots ajoutés. À noter que supprimer un mot de cette liste et redémarrer l'éditeur permet de restaurer sa détection par l'éditeur : on peut ajouter le mot «confusant» (qui n'existe pas) au dictionnaire pour empêcher sa mise en évidence par l'éditeur, mais supprimer le mot du dictionnaire, sauvegarder le fichier DIC et redémarrer l'éditeur fera tout de même apparaître cette mise en évidence. On peut ainsi déduire qu'en réalité le premier dictionnaire n'a rien d'immuable, on pourrait même en supprimer la totalité des mots pour l'expérience, mais compte-tenu du fait qu'une écrasante majorité des utilisateurs ne sait rien des dictionnaires ni de leur localisation, seul le second dictionnaire sera modifié au cours de l'utilisation de l'éditeur de texte, d'où cette qualification.

On connaît maintenant les dictionnaires et leur usage. Mais cela ne permet pas encore de déterminer si le mot actuellement analysé s'y trouve. Il faut pouvoir les parcourir, ce qui pourrait se faire de plusieurs manières.

La première, et la plus intuitive, serait de chercher linéairement dans le premier dictionnaire, en comparant simplement le mot cherché avec tous ceux qu'il contient. Si le mot n'est pas dans le premier dictionnaire, faire de même avec le second. Voici en page suivante ce à quoi pourrait ressembler cette fonction de recherche en Java si l'on définit motATrouver comme le mot dont on vérifie l'écriture, dicoFR comme le dictionnaire principal et dicoPerso comme celui auquel on peut ajouter les éléments. [6]

```java
public static boolean existe(String motATrouver, String[] dicoFR, String[] dicoPerso) {
    for(int i=0 ; i < dicoFR.length ; i++) {
        if(dicoFR[i] == motATrouver) {
            return true ;
        }
    }
    for(int j=0 ; j < dicoPerso.length ; j++) {
        if(dicoPerso[j] == motATrouver) {
            return true ;
        }
    }
    return false;
}
```

Figure 4: Exemple de fonction de recherche linéaire dans les dictionnaires [6]

Cette solution est d'une fiabilité totale, car si on atteint la fin du second dictionnaire, cela signifie que le mot que l'on cherche n'est dans aucun dictionnaire, est donc inconnu et considéré comme mal écrit. L'éditeur de texte peut alors le mettre en évidence. Cependant un point faible demeure ; la complexité. La taille du dictionnaire principal français de LibreOffice est de 84 000 lignes, et celle du dictionnaire personnalisable est soumise à la volonté de l'utilisateur, qui peut très bien y ajouter 30 000 mots s'il le souhaite, et donc 30 000 nouvelles lignes parmi lesquelles chercher. Dans ce cas, si l'on donne à l'éditeur une phrase remplie de mots commençant par la lettre « z », telle que « Zazie zézaie à Zanzibar », alors après segmentation, il faudra parcourir la quasi-totalité du dictionnaire principal pour déterminer quels mots parmi cette phrase existent. De même, si on lui donne une phrase dont tous les mots ont été ajoutés par l'utilisateur, il faudra même aller chercher dans le second dictionnaire chacun des mots. On peut même définir le pire cas de cette méthode, qui serait qu'aucun des mots du texte ne soit dans les dictionnaires. On aurait parcouru les dictionnaires autant de fois sans rien trouver, ce qui aurait un impact considérable sur le temps d'exécution. Ceci nous donne le calcul de complexité suivant. Si le texte à analyser est de longueur L, et le nombre de mots dans les deux dictionnaires égal à N, alors on obtient une complexité équivalente à O(N x L). Cette solution n'est pas viable pour notre problème, car pas assez efficace pour de grands dictionnaires.

Une seconde méthode envisageable fait appel à une structure largement utilisée pour faciliter la recherche de mots dans une liste, il s'agit des arbres radix, parfois appelés arbres PATRICIA. Ils ont l'avantage de grandement faciliter la recherche tout en étant parfaitement fiables. Cependant, en contrepartie, ils sont plus gourmands en stockage qu'une simple liste, en raison de leur structure que l'on va détailler ici, dans le cas d'un arbre adapté pour notre recherche.
Au sommet de cet arbre se trouve la racine. C'est un nœud vide qui contient autant de branches qu'il y a de caractères en première position dans tous les mots des dictionnaires. Il y a donc au moins 26 branches qui descendent de la racine. Il est important de préciser « au moins » car dans des langues sans accents telles que l'anglais, il y aura bel et bien 26 branches, mais il peut y en avoir plus si certains mots de la langue commencent par des caractères accentués. Par exemple en français, avec des mots comme « étable » ou « âcre ». Certains alphabets ont même des caractères présents ni en français, ni en anglais, comme le grec. Un arbre radix a la propriété d'être préfixe. Cela signifie que pour chaque nœud, aucune des branches ne commencera par les mêmes lettres. Pour en faciliter la compréhension, nous avons réalisé un exemple d'arbre radix inspiré d'une illustration à l'origine en anglais.[7] Bien évidemment, une énorme quantité de

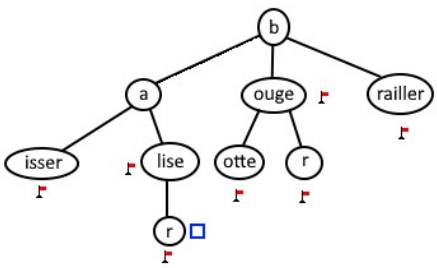

Figure 5: Exemple d'arbre radix

branches n'est pas représentée ici pour maintenir une lisibilité correcte, car il existe sans doute plus de 1000 mots qui commencent par un « b ».
On observe sur l'illustration de petits drapeaux (« flags ») à certains endroits.[7] Ces endroits symbolisent au programme que le nœud actuel est une fin de mot présent dans les dictionnaires. Ainsi, le nœud « a » n'a pas de drapeau car « ba » n'existe pas dans les dictionnaires, alors que « balise », « brailler » et « bougeotte » en ont un car ils s'y trouvent. L'ajout du mot « ba » dans le dictionnaire par l'utilisateur ajouterait un drapeau au niveau du « a ». Plus généralement, chaque nœud qui n'a aucun branche inférieure est marqué d'un drapeau car il est une fin de mot. Cette fonctionnalité des drapeaux améliore la performance : le programme n'a pas à vérifier si le mot de l'arbre où il se trouve actuellement correspond au mot qu'il cherche tant qu'il n'est pas sur un nœud qui porte un drapeau.
Déterminer si un mot est dans les dictionnaire est donc facile pour l'éditeur. Il a juste a suivre le chemin tracé par les branches. Si les branches le mènent au même mot que celui qu'il cherche, alors le mot existe. En revanche, s'il arrive à un nœud qui ne porte pas de drapeau ou qui en porte un mais ne correspond pas, et qu'aucune de ses branches ne permet de suivre le mot recherché, alors il n'existe pas. Dans notre illustration plus haut, le mot « bouger » existe, mais ni le mot « bougef », ni le mot « bak » n'existent, même si dans le premier cas, l'éditeur s'était arrêté sur le mot « bouge » qui existe bien car marqué d'un drapeau.

Si un nouveau mot est ajouté au dictionnaire par l'utilisateur, l'éditeur doit parcourir l'arbre pour déterminer à quel nœud il doit créer une branche correspondant au nouveau mot. Si l'utilisateur parvient à supprimer un mot du dictionnaire, alors l'éditeur a simplement à se rendre au nœud correspondant au mot, et supprimer ce nœud et la branche qui le reliait au précédent. Dans notre arbre, pour ajouter le mot « bougeons », l'éditeur se rendra au nœud « ouge » et créera une branche pour mener au nouveau nœud « ons ». De même, pour supprimer « baliser », l'éditeur se rendra au nœud « r » marqué d'un rectangle bleu (purement indicatif), et supprimera le nœud et la branche qui le reliait au nœud « lise ». Cette méthode d'ajout et de suppression est plus compliquée que celle permise par le fichier texte où il suffit d'ajouter le mot au bon endroit ou le supprimer. Néanmoins, la performance gagnée par l'accélération de la recherche, qui est beaucoup plus souvent utilisée par l'éditeur que l'ajout ou la suppression d'un mot, compense largement ce défaut.

Ainsi, que ce soit par l'utilisation de la liste ou le parcours de l'arbre, l'éditeur de texte est capable, à partir d'un texte, d'en déterminer les mots qui n'existent pas ou lui sont inconnus, et de le signaler à l'utilisateur. C'est cela qu'on appelle la détection d'erreur.

Nous allons à présent traiter une autre fonctionnalité majeure des éditeurs de texte, la correction orthographique. Il s'agit de modifier un mot une fois qu'il a été détecté comme mal écrit lors de la détection d'erreur. Dans cette section, nous verrons les deux types de correction qui existent et comment elles fonctionnent. Pour les différencier, nous allons les nommer par la principale caractéristique qui les distingue : l'une est passive, tandis que l'autre est active.

La correction orthographique passive est qualifiée comme telle car est opérée de manière indépendante par l'éditeur de texte. Elle survient lorsqu'un mot identifié comme une erreur par la détection est inscrit dans les options d'autocorrection. Dans ce cas, l'éditeur remplace simplement le mot originel par ce qui est inscrit dans l'option d'autocorrection. On peut illustrer ce processus par une simple table d'association dans laquelle les clés sont les mots qui seront mal écrits et repérés comme tels par l'éditeur, et les valeurs les mots qui les remplaceront.
L'accès à cette table d'association est facile depuis l'interface de l'éditeur LibreOffice. Il suffit d'aller dans Outils puis Autocorrection et enfin Options d'Autocorrection. On a alors accès, dans de nombreuses langues, à toutes les corrections passives qui sont inclues à l'éditeur par défaut. Parmi

elles, des raccourcis pour écrire des symboles, comme la clé « :alpha: » qui correspond à la valeur du symbole α. On y trouve aussi des erreurs fréquemment commises par les utilisateurs qui sont corrigées au moment où il appuie sur la barre espace. L'utilisateur a même la liberté d'ajouter ses propres autocorrections par cette interface. Ces dernières sont stockées dans des fichiers alloués à LibreOffice sur l'ordinateur, on peut les retrouver en suivant le chemin Windows/Programmes/LibreOffice/share/autocorr.[8] Les fichiers, sous format .dat (pour data), ne s'ouvrent pas directement. Cependant, si l'on copie le fichier acor_fr.dat, qui contient les règles d'autocorrection en français, qu'on le renomme en acor_fr.zip, on peut en extraire le fichier DocumentList.xml. Ce fichier peut être ouvert avec un simple bloc-notes et est un petit peu plus parlant, un extrait en est visible ci-dessous à côté de l'ajout une association d'autocorrection.

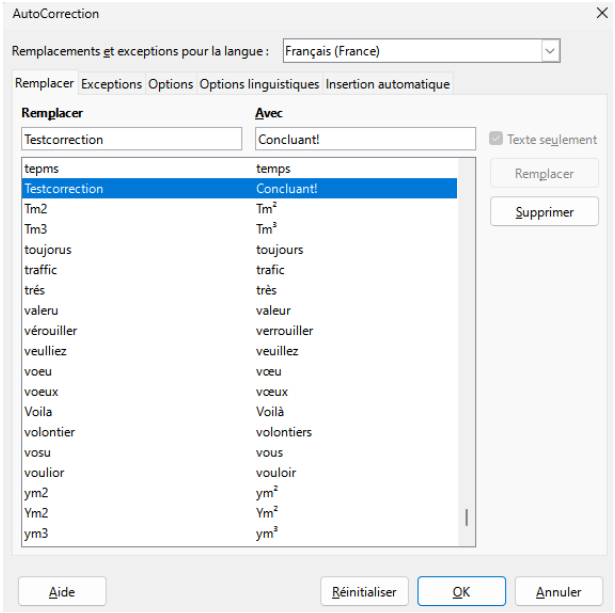

Figure 6: Ajout d'une association dans les options d'autocorrection

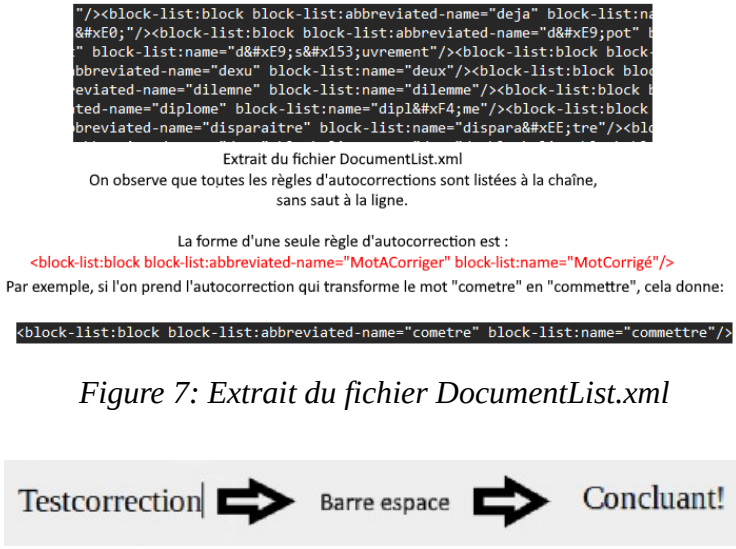

Figure 7: Extrait du fichier DocumentList.xml

Note : Notre règle d'autocorrection veut que le mot « Testcorrection », chaque fois qu'il est détecté par l'éditeur, soit remplacé par le mot « Concluant ! ».

Chaque langue a un fichier de taille différente étant donné que chaque langue a des erreurs fréquemment commises différentes. Par ailleurs, il faut souligner que parfois il est question de praticité : la plupart des gens savent que le mot « œuf » s'écrit avec le caractère « œ » et non « oe ». Cependant, il est laborieux d'aller chercher ce caractère dans les caractères spéciaux. Le correcteur orthographique passif le fait à la place de l'utilisateur après qu'il a écrit « oeuf », ce qui améliore son confort d'utilisation.

Intéressons-nous à présent à la seconde forme de correction. Plus tôt, nous avons évoqué la capacité d'ajout au dictionnaire d'un mot considéré comme mal orthographié par l'éditeur LibreOffice. Il suffisait de faire un clic-droit sur le mot souligné en rouge et de cliquer sur « Ajouter au dictionnaire », comme dans la figure 3. Nous n'avions alors rien dit concernant les mots qui précédaient cette option dans la liste. Ce sont des mots qui, si on s'y intéresse, ont deux propriétés intéressantes : ils ont une écriture proche du mot mal écrit, qui était « commète », et ils sont tous présents dans les dictionnaires. Si l'on avait cliqué sur l'un de ces mots suggérés, alors notre « commète » aurait été remplacée par celui-ci et le soulignage rouge caractéristique d'une erreur

aurait disparu. Cette correction manuelle du texte par l'utilisateur s'appelle la correction active, car elle requiert une action de la part de ce dernier. Cependant, « commète » n'est pas le seul mot que l'utilisateur devra corriger, il faut donc adapter les mots proposés à chaque nouvelle erreur, afin que « ordinnateur » n'obtienne pas les mêmes mots que « commète » en suggestion, ils seraient inappropriés.

Aucun éditeur de texte n'a exposé clairement sa méthode ou son programme pour réaliser ce processus. On va donc essayer de le faire nous-même.
Pour déterminer quels mots suggérer, on dispose d'un outil, la distance de Levenshtein. Cette distance prend une valeur entière, et elle symbolise le nombre d'opérations qu'il est nécessaire de faire, au minimum, pour passer d'un mot A à un mot B. Il y a trois opérations autorisées. Ce sont l'ajout d'un caractère, la suppression d'un caractère, et enfin le changement d'un caractère en un autre. C'est tout. Toutes ces opérations ont une valeur de 1 dans le calcul de la distance. En voici un exemple parlant.

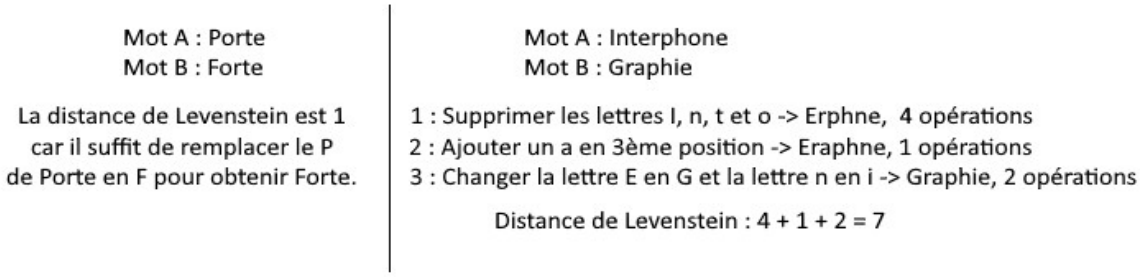

Figure 8: Exemple de calcul d'une distance de Levenshtein

L'éditeur doit pouvoir proposer des mots cohérent. Par exemple, après la détection du mot « bponjour » comme une erreur, il est plus logique de proposer le mot « bonjour » à l'utilisateur que le mot « enclumes », car « bonjour » et « bponjour » ont une distance de Levenshtein de 1, plus petite que celle entre « bponjour » et « enclumes », qui est 8.

Il demeure un problème. Le cerveau humain sait calculer la distance de Levenshtein, mais de façon laborieuse. Or, il faut calculer, pour chaque mot erroné, une distance de Levenshtein avec tous les mots présents dans les dictionnaires, en ne connaissant ni les futurs mots erronés ni les mots qui seront ajoutés au dictionnaire. Il faut donc automatiser le calcul pour qu'il soit fait par l'éditeur, car il aura toujours accès au mot erroné et à ceux du dictionnaire. Le calcul doit être rapide pour que la suggestion des mots le soit aussi.

Pour effectuer le calcul, il existe une méthode qui est très largement inspirée de l'algorithme de Wagner-Fischer [9][10][11], mais nous l'avons légèrement modifiée pour faciliter sa compréhension et surtout à terme pouvoir l'implémenter dans un programme informatique conforme à notre niveau, sur un langage que l'on connaît, c'est-à-dire en Java, et sans récursivité trop poussée. La méthode peut sembler un petit peu compliquée au début, mais elle est en réalité assez facile à prendre en main et surtout très efficace. Elle fait appel à une structure mathématique, la matrice. Il en faudra deux pour chaque calcul de distance. Prenons un mot A de longueur L1 et un mot B de longueur L2, et calculons leur distance de Levenshtein.

La **première matrice** est de taille L1 x L2. Il y a autant de colonnes que de caractères dans L1 et autant de lignes que de caractères dans L2. Le i-ème caractère de L1 est sur la ligne numéro i, et de même pour L2. Chaque case de cette matrice ne peut prendre que deux valeurs : 0 ou 1. Il s'agit de comparer le caractère de la ligne à celui de la colonne. La valeur de la case sera 0 si les caractères

sont les mêmes, 1 sinon. La figure 9 ci-dessous est un exemple dans le cas où le mot A est Graphie, et le mot B est Interphone (L1 = 7 et L2 = 10).

	I	N	T	E	R	P	H	O	N	E
G	1	1	1	1	1	1	1	1	1	1
R	1	1	1	1	0	1	1	1	1	1
A	1	1	1	1	1	1	1	1	1	1
P	1	1	1	1	1	0	1	1	1	1
H	1	1	1	1	1	1	0	1	1	1
I	0	1	1	1	1	1	1	1	1	1
E	1	1	1	0	1	1	1	1	1	0

Figure 9: Exemple de la première matrice

Les zéros, présents dans les cases où les lettres de la ligne et de la colonne sont les mêmes sont en rouge.
La première matrice étant construite, on peut passer à la **seconde**. Elle comporte une ligne et une colonne de plus que la précédente. Sa première ligne ainsi que sa première colonne sont similaires dans leur construction : la première ligne, longue de L2+1 éléments, contient un chiffre par case, de 0 à 10, dans l'ordre croissant. La première colonne est semblable verticalement. Elle contient un chiffre par case de 0 à 7. Voici en figure 10 à quoi doit ressembler la matrice à ce stade, avec les lettres pour faciliter la compréhension.

		I	N	T	E	R	P	H	O	N	E
	0	1	2	3	4	5	6	7	8	9	10
G	1										
R	2										
A	3										
P	4										
H	5										
I	6										
E	7										

Figure 10: Seconde matrice à remplir

L'objectif est de remplir cette matrice. Pour cela, il faut réaliser des paquets de 2x2 éléments dont le nombre inconnu est situé en bas à droite du paquet. On va donc nécessairement aller de gauche à droite et de haut en bas pour remplir la matrice. Sur la figure 10, le premier paquet est représenté en rouge, et le second en bleu. La case dorée, tout en bas à droite de la matrice, sera la valeur finale de la distance de Levenshtein. Il nous faudra forcément remplir toute la matrice pour l'obtenir. Mais avant de pouvoir le faire, il reste une ultime manipulation à effectuer : il faut « superposer » la première et la seconde matrice, afin de faciliter les calculs qui viendront ensuite. Cependant, il faut le faire d'une manière précise, qui va créer un décalage. La figure ci-dessous illustre comment.

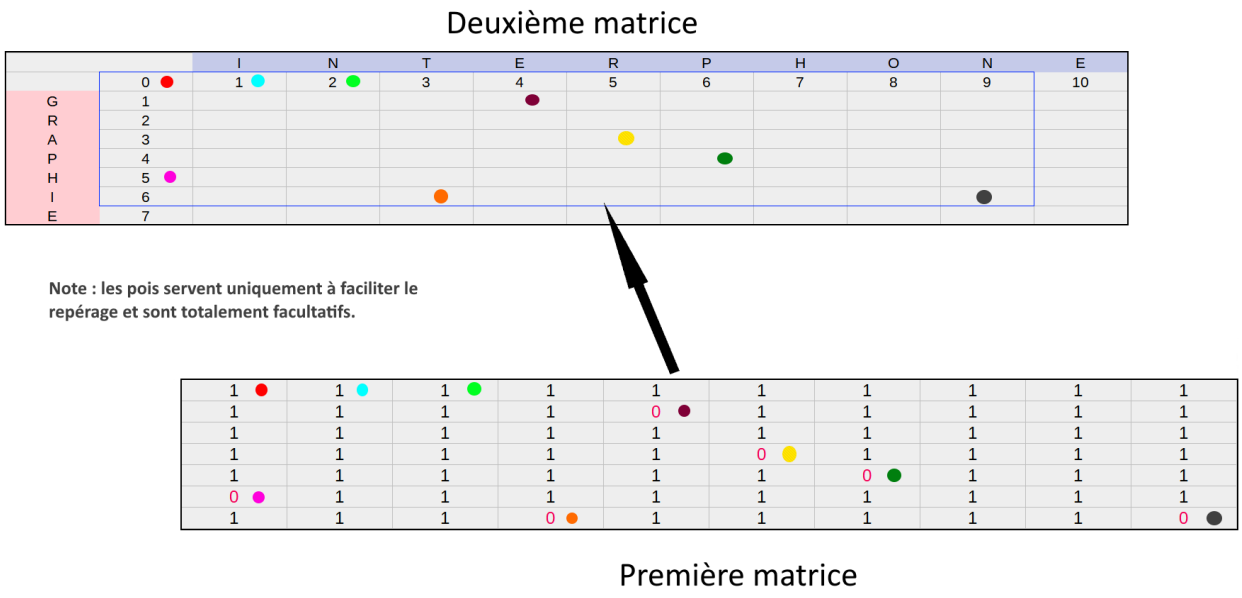

Figure 11: Superposition des deux matrices

Une fois que cette étape est terminée, on peut commencer à remplir les cases vides. On va détailler le raisonnement pour les paquets rouge et bleu de la figure 10 en guise d'exemple. On rappelle qu'on avait déterminé dans la figure 8 que la distance de Levenshtein entre les mots « Interphone » et « Graphie » était 7.

Paquet rouge :
La case que l'on calcule actuellement est la distance de Levenshtein entre les chaînes « G » et « I ».
On va calculer 3 valeurs, la case vide prendra la plus petite parmi celles-ci.
Les 3 valeurs sont :
1 : la valeur de la case juste au dessus de celle qu'on calcule + 1
Cette valeur simule un effacement de caractère.
2 : la valeur de la case juste à gauche de celle qu'on calcule + 1
Cette valeur simule une insertion.
3 : la valeur de la case au dessus à gauche de celle qu'on calcule + la valeur de cette même case dans la matrice superposée, qui est soit 0, soit 1.
Cette valeur simule une substitution, le changement d'un seul caractère en un autre.[12]
Pour faciliter la compréhension de cette dernière valeur, il faut se référer au point rouge de la figure 11. En l'occurrence, la valeur de cette case dans la matrice superposée est 1.
Après calcul, on déduit les 3 valeurs :
1 : 2 (1+1)
2 : 2 (1+1)
3 : 1 (0+1 (point rouge))

On ne conserve que la plus petite valeur, qui est 1. On déduit que cette case contient la valeur 1, et on peut le vérifier : est-ce que la distance de Levenshtein entre les chaînes « G » et « I » est 1 ? Oui, car il suffit de changer le « G » en « I » tout simplement.

Calculons maintenant la case vide du paquet bleu, en guise de second exemple.
Les valeurs sont, à nouveau :
- la valeur de la case juste au dessus de celle qu'on calcule, qui est 2 (en dessous du « N ») à laquelle on ajoute 1
- la valeur de la case juste à gauche de celle qu'on calcule, on vient de l'obtenir, c'est 1, à laquelle on ajoute aussi 1
- la valeur de la case juste au dessus à gauche de celle qu'on calcule, qui est 1, à laquelle on ajoute 1 car c'est sa valeur dans la matrice superposée (pois cyan)

On obtient les valeurs 3, 2, 2. Le minimum est 2. La distance de Levenshtein entre les chaînes « G » et « IN » serait donc 2, ce qui est le cas, il n'y a qu'à ajouter le « I » et changer le « G » en « N ».

Après application de cette méthode pour la matrice entière, on obtient celle visible sur la figure ci-dessous.

		I	N	T	E	R	P	H	O	N	E
	0	1	2	3	4	5	6	7	8	9	10
G	1	1	2	3	4	5	6	7	8	9	10
R	2	2	2	3	4	4	5	6	7	8	9
A	3	3	3	3	4	5	5	6	7	8	9
P	4	4	4	4	4	5	5	6	7	8	9
H	5	5	5	5	5	5	6	5	6	7	8
I	6	5	6	6	6	6	6	6	6	7	8
E	7	6	6	7	6	7	7	7	7	7	7

Figure 12: Seconde matrice remplie

La valeur obtenue dans la case tout à droite étant la distance de Levenshtein entre les chaînes « Graphie » et « Interphone », elle doit être égale à 7 pour être cohérente, ce qui est le cas ici.

Maintenant que l'on sait comment calculer efficacement la distance de Levenshtein sans devoir interpréter le mot comme le ferait un humain, on peut essayer d'implémenter tout ce processus. En effet, malgré plusieurs tentatives de recherche, aucun éditeur n'a clairement exposé son algorithme de calcul de distance de Levenshtein. On peut néanmoins essayer de s'en approcher par nous-même. Notre programme doit pouvoir, avec un mot donné, déterminer tous ceux des dictionnaires qui ont une distance de Levenshtein suffisamment faible avec celui-ci. On peut se fixer une limite et se dire que deux mots sont proches si leur distance de Levenshtein est inférieure ou égale à 2.
Réitérons le processus en l'automatisant.
Il faut d'abord construire la première matrice. Ses dimensions sont les longueurs des mots dont on veut calculer la distance de Levenshtein. Le code Java ci-dessous permet sa construction.

```java
public static int [][] premiereMatrice(String m1, String m2){
    int[][]t1 = new int [m1.length()] [m2.length()];     //on donne à la matrice les bonnes dimensions
    for(int i=0 ; i< m1.length(); i++) {                 //on parcourt la matrice
        for(int j=0 ; j<m2.length(); j++) {
            if(m1.charAt(i)==m2.charAt(j)) {             //si les deux éléments du tableau sont les mêmes
                t1[i][j] = 0;                            //la case prend la valeur 0
            }
            else {
                t1[i][j] = 1;                            //sinon elle prend la valeur 1
            }
        }
    }
    return t1;
}
```

```
1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 0 1 1 1
0 1 1 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 0
```

Après exécution

Figure 13: Création de la première matrice [6]

Notez que la chaîne m1 correspond au mot « Graphie » et la chaîne m2 au mot « Interphone ».
La matrice obtenue, à droite, est représentée par un tableau à deux dimensions et est semblable à celle de la figure 9. Il est maintenant possible de construire la seconde matrice avec la fonction suivante.

```java
public static int[][] distLeven(String m1, String m2) {
    int [] [] premMat = premiereMatrice(m1, m2);
    int [][] t = new int [m1.length()+1] [m2.length()+1];   //initialisation de la matrice aux bonnes dimensions
    for(int i=0 ; i<m1.length()+1 ; i++) {                  //remplissage première ligne
        t[i][0] = i;
    }
    for(int j=1 ; j<m2.length()+1 ; j++) {                  //remplissage première colonne
        t[0][j] = j;
    }

    for(int k = 1 ; k<t.length ; k++) {                     //On positionne le début de la boucle à la première
        for(int l = 1 ; l < t[0].length ; l++) {            //case dont on cherche la valeur
            int valSup = t[k-1][l] + 1;                     //On calcule les 3 valeurs adjacentes
            int valLat = t[k][l-1] + 1;                     //(supérieure, latérale, diagonale)
            int valDiag = t[k-1][l-1] + premMat[k-1][l-1];
            if(valSup <= valLat && valSup <= valDiag) {     //Si la valeur du dessus est la + petite
                t[k][l] = valSup ;                          //alors on la garde
            }
            if(valLat <= valSup && valLat <= valDiag) {     //Si la valeur latérale est la + petite
                t[k][l] = valLat ;                          //alors on la garde
            }
            if(valDiag <= valLat && valDiag <= valSup) {    //Sinon c'est forcément celle en diagonale
                t[k][l] = valDiag;
            }
        }
    }
    return t;

}
```

```
0 1 2 3 4 5 6 7 8 9 10
1 1 2 3 4 5 6 7 8 9 10
2 2 2 3 4 4 5 6 7 8 9
3 3 3 3 4 5 5 6 7 8 9
4 4 4 4 4 5 5 6 7 8 9
5 5 5 5 5 5 6 5 6 7 8
6 5 6 6 6 6 6 6 6 7 8
7 6 6 7 6 7 7 7 7 7 7
```

Figure 14: Création et remplissage de la seconde matrice [6]

On obtient bien la matrice en figure 13, signe que le programme est fonctionnel. Cependant le programme n'est pas encore utilisable par l'éditeur. Maintenant, il faut pouvoir, pour un mot donné, utiliser le programme pour chaque mot des dictionnaires, et stocker les mots dont la distance obtenue est inférieure à 3 dans un tableau. On pourra ensuite proposer les mots de ce tableau à l'utilisateur, qui pourra choisir parmi ces mots existants pour remplacer celui qui est considéré comme erroné.

Pour ce faire, on implémente une fonction basique qui se contente de garder uniquement la valeur en bas à droite de la seconde matrice : c'est la distance de Levenshtein, tout ce dont nous avons besoin pour comparer les mots. Maintenant, il faut pouvoir comparer le mot avec ceux du

dictionnaires. Pour cela, on crée une copie des dictionnaires, dans laquelle on regroupe tous les mots qui les composent, mais en minuscule : en effet, les caractères « A » et « a » ne sont pas les mêmes. « iNtErPhOnE » et « Interphone » auront donc une distance de Levenshtein élevée alors qu'il serait judicieux de suggérer ce dernier. Mettre les mots en minuscule permet de passer outre ce problème. Au final, en plaçant dans un tableau chaque mot du dictionnaire qui a une distance de Levenshtein inférieure ou égale à 2 avec notre fameux mot erroné, on peut établir une liste de mots que l'on pourrait suggérer à l'utilisateur. Voici, sur les deux dernières figures suivantes, la dernière fonction et ce qu'elle renvoie sur trois exemples en guise de test.

```java
public static String [] correcOrtho(String ml) {
    int compteurMotsValides = 0;
    int [] tabIndicesMot = new int [1000];
    int posTabIndice = 0;
    for(int i=0 ; i<dicoFinal.length ; i++) {
        if(returnLev(ml, dicoFinal[i]) <= 2) {  //distance de Levenshtein autorisée fixée à 2 ici
            compteurMotsValides++;
            tabIndicesMot[posTabIndice] = i;
            posTabIndice++;
        }
    }
    if(compteurMotsValides != 0) {
        String [] tabFinal = new String[compteurMotsValides];
        for(int k=0 ; k < tabFinal.length ; k++) {
            tabFinal[k] = dicoFinal[tabIndicesMot[k]];
        }
        return tabFinal;
    }
    else {
        String [] echecRech = new String [1];
        echecRech[0] = "Aucune correction trouvée.";
        return echecRech;
    }
}
```

Figure 15: Fonction finale de génération du tableau de suggestions [6]

```java
String [] Liste = correcOrtho("interohone");
```

Cas 1 : l'utilisateur s'est trompé et a écrit «interohone» au lieu d'«interphone».

Le programme suggère les mots :

```
interphone
interphonie
```

⇦

```java
String [] Liste = correcOrtho("grzphie");
```

Cas 2 : l'utilisateur s'est trompé et a écrit «grzphie» au lieu de «graphie».

Le programme suggère les mots :

```
agraphie
graphie
graphite
gryphée
```

⇦

```java
String [] Liste = correcOrtho("vazrvgizarv");
```

Cas 3 : l'utilisateur écrit n'importe quoi.

Le programme ne trouve aucun mot valide à suggérer, il le fait savoir :

```
Aucune correction trouvée.
```

Figure 16: Exemples de suggestions de correction proposées par le programme

Nous avons réussi à simuler une proposition de corrections à la suite d'une détection d'erreur par l'éditeur de texte. Chaque mot considéré comme tel par l'éditeur peut ainsi être comparé à tous les mots des dictionnaires et suggéré à l'utilisateur si leur distance de Levenshtein est inférieure ou égale à 2. Cette suggestion pourrait se faire par une interface graphique comme celle visible en figure 3, mais nous n'avons pas encore les compétences informatiques pour la réaliser, pour le moment du moins.
Il faut toutefois garder à l'esprit que les correcteurs des gros éditeurs sont certainement plus perfectionnés que celui que nous venons de construire. Par exemple, ils pourraient mettre en place une prise en compte des touches voisines sur le clavier de celle présumée erronée pour adapter les mots suggérés. Néanmoins nous n'avons pas ou que très peu trouvé de documentation sur ce sujet, les éditeurs voulant en général garder privés leurs programmes de détection. Ceci restera donc une hypothèse, que l'on pourrait à l'avenir creuser pour améliorer notre correction orthographique, en impliquant pourquoi pas l'intelligence artificielle, aussi gros ce mot soit-il, à la suggestion de mots.

À l'ère du tout numérique, la vérification et la correction orthographiques sont devenues des outils indispensables pour beaucoup d'utilisateurs. Elles ont permis de détecter puis d'éliminer la plupart des erreurs courantes d'orthographe, de grammaire et de conjugaison dans les textes, et de ce fait, communiquer de manière plus claire et plus efficace. Toutefois, il est important de rappeler que les correcteurs ne sont pas parfaits et peuvent parfois proposer des corrections inadaptées. Il est donc important de les utiliser avec discernement et de toujours se relire. En somme, les outils d'aide à l'écriture sont un outil précieux, mais ne remplaceront jamais l'œil humain à la perfection, tant pour la détection que la correction des erreurs.

Bibliographie :

[1] : « Design and Implementation of Spell Checker for Kashmiri », A. Lawaye et B. S. Purkayastha, juillet 2016,
Disponible à l'adresse :
https://www.researchgate.net/publication/321906322_Design_and_Implementation_of_Spell_Checker_for_Kashmiri

[2] : « Syntactic Wordclass Tagging », H. van Halteren, 1999, page 121
Disponible à l'adresse :
https://books.google.fr/books?id=6SmOssWntJMC&pg=PA121&dq=%22tokenizer%22&hl=fr&sa=X&ei=h86nULzIC82BhQfuzIDYDA&ved=0CDcQ6AEwAQ#v=onepage&q=%22tokenizer%22&f=false

[3] : Source et image réalisée via le site https://www.w3schools.com/python/trypython.asp?filename=demo_ref_string_split

[4] : https://www.linuxquestions.org/questions/ubuntu-63/where-is-the-libreoffice-dictionary-file-located-4175689427/

[5] : https://wiki.documentfoundation.org/UserProfile
Ce ne sont pas des sources d'informations scientifiques mais ce sont les sites sur lesquels on a pu trouver les chemin d'accès aux différents dictionnaires et que l'on tenait donc à mentionner.

[6] : Cette fonction a été réalisée grâce à l'IDE Eclipse, cela nous semblait important de l'indiquer.
Site Web : https://www.eclipse.org/

[7] : « Radix Tree », E. Z. Booker.
Disponible à l'adresse : https://iq.opengenus.org/radix-tree/

[8] : Une fois de plus, c'est une indication de localisation de fichiers qui nous a été donnée et que nous voulons indiquer.
Disponible sur le site : https://ask.libreoffice.org/t/where-are-the-autocorrect-options-saved-in-writer/48434

[9] : https://fr.wikipedia.org/wiki/Distance_de_Levenshtein
[10] : https://fr.wikipedia.org/wiki/Algorithme_de_Wagner-Fischer
[11] : « The String-to-String Correction Problem », R. A. Wagner et M. J. Fischer, 1947.
Disponible à l'adresse : https://dl.acm.org/doi/pdf/10.1145/321796.321811

[12] : « Levenshtein Distance Computation », S. Grashchenko, mis à jour le 16 novembre 2022, section 5.2.
Disponible à l'adresse :https://www.baeldung.com/cs/levenshtein-distance-computation