Composition filtrée et marqueurs de règles de réécriture pour une distance d'édition flexible

Application à la correction des mots hors vocabulaire

Richard Beaufort

Centre de traitement automatique du langage
Université de Louvain
Place Blaise Pascal 1, 1348 Louvain-la-Neuve, Belgique
richard.beaufort@uclouvain.be

RÉSUMÉ. *Nous présentons une implémentation flexible et originale de la distance d'édition : la composition filtrée, un type particulier de composition de deux machines à états finis au travers d'un filtre qui modélise l'ensemble des opérations d'édition valides. Le filtre est un transducteur pondéré ou une cascade de transducteurs pondérés. Il est obtenu par compilation de règles de réécriture qui profitent d'un nouveau concept défini dans notre bibliothèque de machines à états finis : le marqueur de règles, un symbole qui n'appartient pas à l'alphabet utilisé, mais est inséré dans une règle de réécriture afin d'identifier un phénomène et d'en suivre l'évolution. Les marqueurs désambiguïsent et facilitent l'expression de conditions et de contraintes. La méthode est illustrée dans le cadre de la correction des mots hors vocabulaire.*

ABSTRACT. *We present an original and flexible implementation of the edit-distance: the filtered composition, a special kind of composition of two finite-state machines through a filter that models all valid edit-operations. The filter is either a weighted transducer or a cascade of weighted transducers. It is built from weighted rewrite rules that take advantage of a new concept defined in our finite-state framework: the rules' marker, a symbol that does not belong to the alphabet in use, but is inserted into a rewrite rule in order to mark a phenomenon and to track its evolution. Markers disambiguate and make it easy to express conditions and constraints. The method is illustrated on the task of correcting out-of-vocabulary words.*

MOTS-CLÉS : *distance d'édition, composition filtrée, marqueurs de règles, correction orthographique.*

KEYWORDS: *edit distance, filtered composition, rules' markers, spelling correction.*

1. Introduction

La distance d'édition $D(u, v)$ de deux séquences u et v mesure le nombre minimal d'opérations d'édition (substitution, insertion, suppression) nécessaires pour convertir u en v. Le principe est très intéressant, mais son implémentation pose de nombreuses questions pratiques auxquelles de nombreuses études ont tenté de répondre. Comment pondérer efficacement les opérations d'édition ? Comment modéliser des substitutions n–m ? Et, enfin et surtout, comment construire rapidement l'ensemble V de séquences v appartenant à un lexique L, pour lesquelles la distance d'édition $D(u, v)$ ne dépasse pas un certain seuil ?

La méthode que nous présentons est une extension de la distance d'édition de deux automates définie par Mohri (2003). Étant donné une forme u et un lexique L, nous proposons de construire l'ensemble V de candidats v pour u dans L au travers de la cascade de compositions suivante :

$$\mathcal{V} = \mathcal{U} \circ \mathcal{F} \circ \mathcal{L} \qquad [1]$$

où $\mathcal{F}$ est un transducteur pondéré qui modélise les opérations d'édition acceptées. La méthode est appelée *composition filtrée*, parce que le transducteur pondéré $\mathcal{F}$ peut être considéré comme un filtre qui détermine la taille de l'intersection entre u et L. Dans le cadre de nos outils à états finis, le filtre est construit à partir de règles de réécriture pondérées qui tirent avantage d'un nouveau concept que nous avons défini : le marqueur de règles, un symbole spécial qui désambiguïse les règles de réécriture et facilite l'expression de conditions et de contraintes.

Cet article s'organise comme suit. La section 2 présente les concepts importants relatifs aux machines à états finis. La section 3 propose ensuite un survol de la distance d'édition dans la littérature. Sur cette base, nous définissons la composition filtrée en section 4 et les marqueurs de règles en section 5. La méthode est ensuite appliquée à la correction des mots hors vocabulaire, dans le contexte de textes dactylographiés en section 6, et dans celui de la reconnaissance optique de caractères en section 7. La section 8, enfin, tire quelques conclusions et suggère quelques perspectives.

2. Les machines à états finis

Le lecteur intéressé consultera utilement Hopcroft *et al.* (1979) et Kuich et Salomaa (1986) pour les fondements du domaine. Nous ne proposons ici qu'un rappel des notions utiles à la suite de notre exposé. Les machines à états finis (FSM, *finite-state machines*) sont des graphes orientés étiquetés. Elles regroupent les automates (FSA, *finite-state automaton*), les transducteurs (FST, *finite-state transducer*) et leurs équivalents pondérés (WFSA, *weighted FSA*, et WFST, *weighted FST*). L'un des atouts majeurs des FSM est la possibilité de leur appliquer des algorithmes d'optimisation (suppression-ϵ, déterminisation, minimisation, projection, etc.) qui assurent une efficacité optimale en termes d'espace de représentation et de temps de traitement (Aho *et al.*, 1986 ; Mohri *et al.*, 2000).

Les FSA, dont les transitions ne sont étiquetées que par un seul symbole, sont les équivalents des langages rationnels (Kleene, 1956) générés par les grammaires de type 3 de la hiérarchie de Chomsky (1956). Cette équivalence est entre autres à la base d'algorithmes permettant de compiler une expression rationnelle en l'automate correspondant (McNaughton et Yamada, 1960).

Les FST, dont les transitions sont étiquetées d'un symbole d'entrée et d'un symbole de sortie, mettent en relation des langages rationnels. Sur cette base, Johnson (1972) a démontré que des ensembles de règles de réécriture de la forme

$$\phi \rightarrow \psi :: \lambda \_ \rho \qquad [2]$$

où ϕ se réécrit ψ dans le contexte de λ et ρ, peuvent être compilées séparément sous la forme de FST distincts, composés ensuite ensemble dans l'ordre des règles, afin d'obtenir un FST unique appliquant l'ensemble de règles en une seule opération. Le principe de composition, qui est une généralisation de l'intersection des automates aux transducteurs (Pereira *et al.*, 1994), respecte naturellement la contrainte exprimée sur les règles de réécriture, qui veut que les règles soient ordonnées de la plus spécifique à la plus générale, de manière à ce qu'une règle ne s'applique sur une entrée que si aucune autre règle plus spécifique n'a été rencontrée précédemment.

Les machines pondérées, dont les transitions et les états peuvent présenter des poids, sont les équivalents des séries rationnelles (Schützenberger, 1961). Elles constituent un choix de premier plan pour résoudre des problèmes du type « recherche des n meilleures séquences dans un treillis de solutions » (Eilenberg, 1974 ; Kuich et Salomaa, 1986). Il est en outre possible de compiler, sous la forme de WFST, des ensembles de règles de réécriture pondérées de la forme

$$\phi \rightarrow \psi :: \lambda \_ \rho \,/\, \omega \qquad [3]$$

où le poids ω est attribué au remplacement $\phi \rightarrow \psi$ (Mohri et Sproat, 1996).

3. La distance d'édition dans la littérature

La distance d'édition $D(u, v)$ de deux séquences u et v mesure le nombre minimal de substitutions, d'insertions et de suppressions nécessaires pour convertir u en v (Damerau, 1964 ; Levenshtein, 1966). Le principe est classiquement mis en œuvre par programmation dynamique (Wagner, 1974), et les systèmes acceptent typiquement un maximum de deux opérations d'édition entre u et v (Yannakoudakis et Fawthrop, 1983 ; Peterson, 1986). Tous les mots v ajoutés à l'ensemble V des candidats sont ensuite pondérés comme suit :

$$p(v|u) = \begin{cases} \alpha & \text{si } v = u \\ \frac{1-\alpha}{|V|-1} & \text{sinon} \end{cases} \qquad [4]$$

où α est un paramètre empirique. Typiquement, $\alpha = 0,99$.

Dans cette implémentation classique, la distance d'édition souffre de deux inconvénients. Premièrement, même avec un algorithme efficace, le calcul séquentiel de la distance d'édition entre le mot u et chaque mot v du lexique n'est plus envisageable dès que le lexique contient quelques milliers de formes. Or, un lexique réaliste contient au moins plusieurs centaines de milliers de formes…
Deuxièmement, la manière dont les candidats sont pondérés n'est pas efficace. Dans V, tous les candidats différents de u sont équiprobables, et plus il y a de candidats, moins il est probable qu'un candidat différent de u soit choisi. En outre, les substitutions n–m, telles que les terminaisons homophones en français (*-er*, *-ez*, *-ées*, etc.) ou les erreurs réalisées par un système de reconnaissance optique des caractères ($h \leftrightarrow li$, $m \leftrightarrow rn$), sont au mieux modélisées sous la forme de plusieurs opérations d'édition, ce qui a pour conséquence d'écarter des candidats pertinents au profit d'autres qui le sont moins. Enfin, le nombre maximal d'erreurs autorisées dans un mot est le même pour tous les mots, indépendamment de leurs longueurs.

Au vu de l'intérêt du principe, cependant, de nombreux travaux se sont concentrés sur l'optimisation de la pondération et/ou de la consultation du dictionnaire.

3.1. *Optimisation de la pondération*

Quelle que soit l'approche, l'idée est d'utiliser un modèle de langue, où chaque forme $v \in V$ est analysée au travers de la métaphore du canal bruité : *étant donné une séquence d'observations u, trouvez quelle séquence voulue v est la plus probable, en maximisant*

$$P(v|u) = P(v)\,P(u|v) \quad [5]$$

où $P(v)$, qui caractérise la source, est une probabilité *a priori* estimée sur un corpus de formes correctes, et $P(u|v)$, qui caractérise la transduction entre la forme corrompue et la forme voulue, est estimé sur un corpus de paires alignées (*forme corrompue*, *forme correcte*). Les approches diffèrent uniquement au niveau des caractéristiques prises en compte dans le calcul de $P(u|v)$. Church et Gale (1991) ont défini $P(u|v)$ comme le produit des opérations d'édition nécessaires pour convertir u en v :

$$P(u|v) = \prod_i ed(u_i, v_i) \quad [6]$$

où $ed(u_i, v_i)$ peut être n'importe quelle opération d'édition classique. Cette première approche, cependant, n'autorise pas les substitutions n–m. Sur cette base, Brill et Moore (2000) ont proposé un modèle dans lequel $P(u|v)$ estime le meilleur alignement de u et v :

$$P(u|v) = \max_{x \in Seg(u), y \in Seg(v)} \prod_{i=1}^{|y|} P(x_i|y_i) \quad [7]$$

où $Seg(u)$ est l'ensemble de toutes les segmentations possibles de u. Cette approche modélise les substitutions n–m, mais ne gère pas les erreurs cognitives. Par exemple, *edelvise* est corrigé par *advise* à la place de *edelweiss*. Ce constat a convaincu Toutanova et Moore (2002) de proposer une approche dans laquelle le modèle graphémique

présenté en équation 7, et que nous référençons $P_{gra}(u|v)$ ci-dessous, est complété par un nouveau modèle phonétique $P_{pho}(u|v)$. Ces deux probabilités sont combinées dans la prise de décision :

$$P(u|v) = P_{gra}(u|v) \quad \lambda\, P_{pho}(u|v) \qquad [8]$$

Dans ce modèle, λ peut être considéré comme un paramètre d'interpolation. Il est entraîné sur un corpus de données extérieures (*held-out data*) aux corpus d'entraînement des modèles qu'il combine. Son estimation a été réalisée par simple recherche en grille (*grid search*), sur la base de quelques valeurs comprises entre 0 et 1. Les substitutions phonétiques, quant à elles, sont estimées :

$$P_{pho}(u|v) = \textstyle\sum_{\bar{v}} \frac{1}{|\bar{v}|} \max_{\bar{u}} \left(P(\bar{u}|\bar{v})\, P(\bar{u}|u)\right) \qquad [9]$$

où $\bar{x}$ note la transcription phonétique de la séquence graphémique x. Dans le cas d'un mot hors vocabulaire (OOV, *out-of-vocabulary word*), dont la catégorie grammaticale est inconnue ou peu fiable, le système propose les trois transcriptions phonétiques *les plus probables*. Ces transcriptions phonétiques sont obtenues à l'aide d'une version modifiée de l'algorithme de Fisher (1999), qui apprend des règles de phonétisation de la forme :

$$L_c \quad \rightarrow \quad Pho \quad :: \quad Gra_g \quad \_ \quad Gra_d \quad / \quad w \qquad [10]$$

où L_c est la lettre cible, Pho est le phonème proposé, Gra_g et Gra_d sont des graphèmes de 1 à 4 lettres situés respectivement à gauche et à droite de L_c, et w est le poids associé à la conversion $L_c \rightarrow Pho$. Ces règles, apprises à partir d'un corpus de paires alignées (*forme lexicale*, *phonétisation*), sont stockées dans une table de hachage dont les clefs sont constituées de la lettre cible et des contextes gauches et droits séparés par des points :

$$Gra_g.L_c.Gra_d \quad \rightsquigarrow \quad \boxed{Pho \quad : \quad w}$$

Chez Toutanova et Moore (2002), l'une des modifications apportées à la méthode de Fisher (1999) consiste à utiliser les cinq règles les plus probables de la table de hachage *pour chaque lettre cible*. Pour un mot de k lettres, le processus est donc susceptible de construire k*5 transcriptions. De cet ensemble de transcriptions, les trois les plus probables seront finalement sélectionnées à l'aide d'un trigramme entraîné sur des séquences phonétiques. Toutanova et Moore (2002) ne donnent aucune indication quant au temps nécessaire à la construction et à la sélection de ces transcriptions phonétiques, mais il est évident que ce processus non déterministe doit augmenter significativement le temps global de la méthode. La phonétisation des OOV en cours de correction pénalise donc le temps de traitement de cette approche qui, au demeurant, est la seule à gérer efficacement les substitutions n–m et à tenir compte des erreurs cognitives.

3.2. *Optimisation de la consultation du dictionnaire*

Dans le but de factoriser les opérations communes aux formes du lexique qui partagent des sous-séquences communes, le formalisme puissant des machines à états finis peut être utilisé. Bien sûr, ceci présuppose que le lexique L soit représenté sous la forme d'une machine à états finis $\mathcal{L}$ (Aho, 1990 ; Mohri, 1996). Sur cette base, Oflazer (1996) a défini un nouveau parcours du FSA $\mathcal{L}$ guidé par la séquence u à reconnaître. L'algorithme utilise une pile de paires (v, p), où v est la séquence en cours de formation et p est l'état courant du FSA. Une séquence v n'est ajoutée à V que si la distance d'édition $D(u, v)$ est en deçà d'un certain seuil max. La caractéristique principale de cet algorithme est que la paire (v, p) n'est ajoutée à la pile que si un test, le *cutoff*, l'autorise. Le cutoff retourne la distance d'édition minimale entre v et certaines sous-séquences de u[1]. Ainsi, ce test gère à la fois la taille de la pile et la profondeur de la recherche.

Cet algorithme est très efficace. Cependant, Schulz et Mihov (2002) ont montré que deux opérations sont très coûteuses : le cutoff, réalisé pour chaque transition, et la distance d'édition, calculée chaque fois qu'un état final est atteint. Afin d'éviter cela, ils ont proposé de construire V au travers d'une simple intersection d'automates :

$$\mathcal{V} = \mathcal{U}_{ed} \cap \mathcal{L} \qquad [11]$$

où $\mathcal{U}_{ed}$ est un FSA, construit à partir de u, qui représente l'ensemble de toutes les séquences v distantes de u en deçà d'un seuil max donné. La méthode, très efficace, obtient des temps de traitement nettement inférieurs à ceux de l'algorithme précédent.

3.3. *Optimisation de l'ensemble du processus*

Les deux approches à états finis précédentes sont efficaces, mais ne gèrent pas les substitutions n–m et implémentent la fonction de coût classique où toute opération d'édition vaut 1. Ce constat est à la base de deux autres approches à états finis. La première (Kolak *et al.*, 2003) calcule la distance d'édition entre u et v comme le coût de leur meilleur alignement, évalué au travers de leurs meilleures segmentations. Cette méthode, appliquée à la sortie d'un système de reconnaissance des caractères (OCR, *optical character recognition*), est fondée sur l'entraînement de deux modèles de correction. Le premier, noté $\mathcal{T}(L{:}Seq)$, découpe chaque mot en ses deux sous-séquences les plus probables :

$$example \Rightarrow \{\, ex \mid ample\,,\ exam \mid ple\}$$

Le second modèle, noté $\mathcal{T}(Seq{:}Seq')$, apprend sur corpus des listes de correspondances entre séquences correctes et séquences corrompues :

$$exam \Rightarrow exain, cxam, \ldots$$

1. Les sous-séquences de longueur l, tel que $\max(1, |v| - max) \leq l \leq \min(|u|, |v| + max)$.

Les deux modèles sont ensuite composés ensemble, et le FST $\mathcal{T}(L{:}Seq')$ qui en résulte est inversé, de manière à obtenir un lexique $\mathcal{T}(Seq'{:}L)$ qui accepte des mots corrompus en entrée et produit des mots corrects en sortie. Le résultat de la composition

$$\mathcal{V} = \mathcal{U} \circ \mathcal{T}(Seq'{:}L) \quad [12]$$

est un FST qui projette u sur un treillis de séquences pondérées v du lexique. Cette approche, qui réduit significativement le taux d'erreurs de l'OCR, présente néanmoins deux inconvénients. D'une part, le modèle de confusion est dépendant du contexte et ne propose pas de remplacement pour des sous-séquences corrompues non apprises à l'entraînement. D'autre part, le modèle de correction est partie intégrante du lexique, ce qui interdit des simplifications intermédiaires (projection, suppression-ϵ, etc.) qui pourraient réduire la place mémoire et le temps de traitement.

La seconde approche (Mohri, 2003) n'est pas dédiée à la recherche d'un mot dans un lexique, mais propose un algorithme plus général, permettant de calculer la distance d'édition de deux automates. Soit $\mathcal{A}_1$ et $\mathcal{A}_2$, deux automates dont on veut connaître la distance d'édition. L'idée est de composer $\mathcal{A}_1$ et $\mathcal{A}_2$ au travers d'un transducteur $\mathcal{F}$ qui représente l'ensemble des opérations d'édition autorisées :

$$\mathcal{L} = \mathcal{A}_1 \circ \mathcal{F} \circ \mathcal{A}_2 \quad [13]$$

La figure 1 donne un exemple de transducteur autorisant les trois opérations d'édition classique associées à un coût constant de 1. Le résultat de la composition, $\mathcal{L}$, est un ensemble de couples de séquences (x, y), chacune associée à un coût w correspondant au nombre d'opérations d'édition nécessaires pour convertir x en y. Plus formellement,

$$\mathcal{L} = \{((x, y), w) : x \in \mathcal{A}_1, y \in \mathcal{A}_2, w \in \mathbb{R}\} \quad [14]$$

où $\mathbb{R}$ est l'ensemble des réels non négatifs[2]. L'un de ces couples, dont le coût w est minimal, correspond à la distance d'édition entre $\mathcal{A}_1$ et $\mathcal{A}_2$, et peut être obtenu par application de n'importe quel algorithme de recherche du plus court chemin d'un graphe (Mohri, 2002) :

$$D(\mathcal{A}_1, \mathcal{A}_2) = \min \mathcal{L} \quad [15]$$

Le transducteur $\mathcal{F}$, qui rend cet algorithme possible, résout les deux inconvénients de la distance d'édition classique : il facilite la pondération des opérations d'édition, et autorise la définition de substitutions n–m.

2. En réalité, il s'agit du semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$. Le lecteur intéressé consultera Mohri (2002).

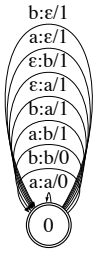


Figure 1. *WFST $\mathcal{F}$, défini sur l'alphabet $\{a, b\}$*

4. La composition filtrée

4.1. *Principe*

La composition filtrée est une extension de la notion de *distance d'édition de deux automates* proposée par Mohri (2003). Avant de présenter le modèle que nous proposons, il faut préciser que la distance d'édition définie sur les automates peut être étendue aux transducteurs et se calcule de manière identique. Soit $\mathcal{T}_1$ et $\mathcal{T}_2$, deux transducteurs. Leur distance d'édition correspond à :

$$D(\mathcal{T}_1, \mathcal{T}_2) = \min(\mathcal{T}_1 \circ \mathcal{F} \circ \mathcal{T}_2) \quad [16]$$

Dans ce cas-ci cependant, le meilleur chemin n'est pas le couple de séquences (x, y) sur lequel a été calculée la distance d'édition, mais le couple de séquences (z_1, z_2) qui leur sont associées respectivement dans $\mathcal{T}_1$ et $\mathcal{T}_2$. Plus formellement,

$$D(\mathcal{T}_1, \mathcal{T}_2) = \min\{((z_1, z_2), w) : (z_1, x) \in \mathcal{T}_1, (y, z_2) \in \mathcal{T}_2, w \in \mathbb{R}\} \quad [17]$$

La composition filtrée n'est pas seulement un moyen de trouver la distance d'édition entre deux langages. Plus généralement, la composition filtrée modifie les possibilités d'intersection entre deux langages, et peut associer un poids à chaque chemin de cette intersection. En outre, comme le mentionne Mohri (2003), tout WFST $\mathcal{F}$ aux poids non négatifs peut être utilisé comme filtre. En particulier, $\mathcal{F}$ peut lui-même être une cascade de transductions pondérées. Par exemple, $\mathcal{F} = \mathcal{F}' \circ \mathcal{F}''$. Nous définissons le filtre sur cette base.

Définition 1 (filtre) *Un filtre est un WFST ou une cascade de WFST dont le rôle est de modifier la taille de l'intersection entre deux langages, et qui peut associer un poids à chaque chemin de cette intersection. Le filtre est permissif lorsqu'il augmente la taille de l'intersection, et restrictif lorsqu'il la réduit.*

Quand un filtre est une cascade de transductions, $\mathcal{F} = \mathcal{F}' \circ \mathcal{F}''$, l'associativité de la composition nous permet de calculer $\mathcal{T}_1 \circ \mathcal{F} \circ \mathcal{T}_2$ de différentes manières :

$$\begin{array}{rclcl} \mathcal{T}_1 \circ \mathcal{F} \circ \mathcal{T}_2 & = & \mathcal{T}_1 \circ (\mathcal{F}' \circ \mathcal{F}'') \circ \mathcal{T}_2 & = & (\mathcal{T}_1 \circ \mathcal{F}') \circ (\mathcal{F}'' \circ \mathcal{T}_2) \\ & = & (\mathcal{T}_1 \circ \mathcal{F}' \circ \mathcal{F}'') \circ \mathcal{T}_2 & = & \mathcal{T}_1 \circ (\mathcal{F}' \circ \mathcal{F}'' \circ \mathcal{T}_2) \\ & = & \ldots & & \end{array}$$

Cette propriété est d'un grand intérêt, parce qu'elle facilite la définition de filtres complexes, mais utilisables. En effet, calculer le WFST $\mathcal{T}$ correspondant à une cascade de transductions peut s'avérer impossible si la taille de $\mathcal{T}$ excède la mémoire disponible. À la place, les parties du filtre peuvent être successivement composées avec la séquence d'entrée u à analyser. Cette approche réduit la taille du filtre aux possibilités de la séquences, et permet de réaliser des étapes d'élagage (projection, suppression-ϵ, etc.) après chaque composition.

4.2. *Modèle*

Soit u, la forme lexicale à analyser, et L, le lexique utilisé. L'ensemble V des candidats v pour u dans L est construit par composition filtrée, au travers d'un filtre permissif F :

$$\mathcal{V} = \mathcal{U} \circ \mathcal{F} \circ \mathcal{L} \qquad [18]$$

Ce modèle nécessite quelques commentaires :

(1) le modèle n'implique pas que u soit hors vocabulaire. S'il l'est, V contiendra uniquement des candidats du lexique. Sinon, u appartiendra à V et sera son propre meilleur candidat ;

(2) $\mathcal{L}$ peut être pondéré. Dans le cas d'un WFSA, chaque forme $v \in L$ sera associée à son maximum de vraisemblance $p(v)$ appris sur corpus. Dans le cas d'un WFST qui projette toute forme lexicale $v \in L$ sur une liste d'analyse morphosyntaxiques $\{t_i\}$, tout couple (v, t_i) sera associé à une probabilité $p(v|t_i)$ apprise sur corpus. L'estimation que nous proposons pour $p(v|t_i)$ est détaillée dans Beaufort *et al.* (2002) ;

(3) $\mathcal{U}$ peut être plus qu'une simple forme lexicale. Cet automate peut être un moyen simple de modéliser un treillis pondéré de formes lexicales. Un exemple de ceci est donné en section 7 dans le cadre d'un système OCR ;

(4) le WFST $\mathcal{F}$ modélise toutes les opérations d'édition valides et est toujours le résultat de la compilation de règles de réécriture pondérées, soit conçues par un expert, soit construites à partir d'un entraînement. Les règles prennent la forme :

$$\phi \rightarrow \psi :: \lambda\_\rho / w \qquad \text{ou} \qquad \phi \,?\!\rightarrow \psi :: \lambda\_\rho / w$$

où appliquer le remplacement obligatoire ($\rightarrow$) ou facultatif ($?\!\rightarrow$) dépend du contexte (λ_ρ) et reçoit le poids w, qui est soit la distance d'édition standard, soit

une valeur empirique, soit, enfin, une probabilité résultant d'un entraînement sur corpus. Le type d'opération réalisée se déduit de la forme du remplacement :
- $a \ (?)\rightarrow \ b$: substitution 1–1 ;
- $\epsilon \ (?)\rightarrow \ a$: insertion ;
- $a \ (?)\rightarrow \ \epsilon$: suppression ;
- $x \ (?)\rightarrow \ y$: substitution n–m ($|x| = n, |y| = m$, avec $n, m \geq 1$).

5. Les marqueurs de règles

Nous l'avons mentionné, le filtre $\mathcal{F}$ est le résultat de la compilation de règles de réécriture pondérées. En théorie, concevoir des règles de réécriture est plutôt aisé, la seule difficulté est de décrire le contexte dans lequel une règle s'applique. Construire un ensemble de règles cohérent, cependant, est délicat. Ceci est illustré dans le tableau ci-dessous, où l'application d'un ensemble de règles (I ou II) donne le même résultat, que la séquence traitée soit *cad* ou *cd*. En fait, le problème est que ces règles sont incapables de faire la différence entre un symbole original et un symbole réécrit.

	Règles I	**Ex.**	*cad*	*cbd*
1.	$a \rightarrow b :: c\_d$		*c*<u>*b*</u>*d*	*cbd*
2.	$b \rightarrow e :: c\_$		*c*<u>*e*</u>*d*	*c*<u>*e*</u>*d*

	Règles II	**Ex.**	*cad*	*cd*
1.	$a \rightarrow \epsilon :: c\_d$		<u>*cd*</u>	*cd*
2.	$c \rightarrow e :: \_d$		<u>*e*</u>*d*	<u>*e*</u>*d*

Une solution élégante à ce problème est d'utiliser des *marqueurs de règles*. La notion de *marqueur*, en soi, n'est pas nouvelle dans le contexte des règles de réécriture. L'usage que nous en faisons, en revanche, est différent : nous l'utilisons pour construire des règles de réécriture non ambiguës, alors que le marqueur est classiquement utilisé dans l'algorithme de compilation qui convertit une règle de réécriture en transducteur à état finis. Nous commençons par présenter notre conception du marqueur en section 5.1 et sa mise en œuvre en section 5.2, avant de la comparer en section 5.3 aux travaux antérieurs du domaine.

5.1. *Le marqueur, comme aide à la construction de règles*

Nous nous inspirons ici de la biologie moléculaire (Miki et McHugh, 2004), où un marqueur de gène est un trait génétique ou un segment d'ADN facilement identifiable, qui est inséré par l'être humain dans un organisme afin de pister l'évolution d'un autre gène, appelé le *gène cible*. Le marqueur de gène doit être sur le même chromosome que le gène cible et suffisamment proche de lui pour que marqueur et cible soient hérités ensemble. Nous donnons au marqueur de règles un sens similaire dans le contexte des règles de réécriture.

Définition 2 (marqueur de règles) *Un marqueur de règles est inséré dans une règle de réécriture par son concepteur, qui désire marquer un phénomène et en suivre l'évolution. Considéré par le compilateur de règles comme un symbole et non comme un*

opérateur, le marqueur est cependant identifiable, parce qu'il n'appartient pas à l'alphabet en usage.

Tel que défini ci-dessus, le marqueur n'est donc pas inséré par le compilateur de règles. Le compilateur est en revanche capable de l'identifier, et le gère comme un symbole extérieur à l'alphabet en usage, qui s'ajoute à celui-ci. Ces précisions sont fondamentales, parce qu'elles distinguent nos marqueurs de ceux des travaux antérieurs (*cf.* section 5.2).

Le marqueur peut remplir trois fonctions : *déclencheur*, *masqueur* ou *bloqueur*. Le déclencheur indique qu'une condition suffisante à l'application d'une règle est remplie. En pratique, une règle signale qu'une condition a été rencontrée en insérant le déclencheur τ correspondant. Les règles suivantes peuvent dès lors subordonner leur application à la présence de τ. Dans l'exemple suivant,

$$\begin{array}{ll}(1) & \epsilon \rightarrow \tau :: [condition] \\ (2) & a \rightarrow b :: \_\,\tau\, def \\ (3) & g\,\tau \rightarrow h :: \_\, def \\ (4) & \tau \rightarrow \epsilon \end{array}$$

la première règle insère le déclencheur τ. La deuxième l'utilise, mais ne le réécrit pas. De ce fait, d'autres règles qui le référencent peuvent encore s'appliquer. La troisième règle, en revanche, réécrit τ, ce qui empêche l'application de toute autre règle qui aurait nécessité sa présence. La dernière règle le supprime, au cas où il serait encore présent.

Le masqueur, lui, cache une expression régulière. Typiquement, un masqueur μ est utilisé en lieu et place du remplacement (ψ) d'une règle. En pratique, une règle remplace la cible (ϕ) par un masqueur μ dédié, et une règle située à la fin de l'ensemble de règles réécrit le masqueur par le remplacement correspondant. Dans l'exemple suivant,

$$\begin{array}{ll} (1) & a \rightarrow b :: c\_d \\ (2) & b \rightarrow e :: c\_d \end{array} \quad \text{devient} \quad \begin{array}{ll} (1) & a \rightarrow \mu :: c\_d \\ (2) & b \rightarrow e :: c\_d \\ (3) & \mu \rightarrow b \end{array}$$

la première règle cache le remplacement b par μ. Ceci empêche la seconde règle de s'appliquer sur une entrée réécrite par la première règle. Située à la fin de l'ensemble, la troisième règle remplace μ par b en toute sécurité.

Le bloqueur est inséré entre deux expressions régulières afin d'empêcher qu'ensemble, ces expressions n'en constituent une nouvelle. Un bloqueur est nécessaire dès qu'une règle supprime sa cible, et qu'aucune règle ne peut s'appliquer sur la nouvelle expression régulière créée par cette suppression. Comme le masqueur, le bloqueur β ne peut être réécrit que lorsque tout risque de réécriture excessive a disparu :

$$\begin{array}{ll} (1) & a \rightarrow \epsilon :: c\_d \\ (2) & c \rightarrow e :: \_\, d \end{array} \quad \text{devient} \quad \begin{array}{ll} (1) & a \rightarrow \beta :: c\_d \\ (2) & c \rightarrow e :: \_\, d \\ (3) & \beta \rightarrow \epsilon \end{array}$$

En somme, le bloqueur peut être considéré comme un masqueur particulier, destiné à cacher une suppression. Conceptuellement, les deux marqueurs sont cependant distincts, étant donné que le masqueur empêche la réécriture du remplacement lui-même, alors que le bloqueur empêche la réécriture du *contexte* qui entoure une suppression.

Cela dit, il est important de mentionner que les distinctions établies entre déclencheur, masqueur et bloqueur ne sont que théoriques. En pratique, le type d'un marqueur n'est jamais spécifié, mais se déduit de l'emploi qui en est fait. Par exemple, dans les règles ci-dessous, le marqueur μ est un masqueur en *I*, mais un déclencheur en *II* :

I (1) $a \rightarrow \mu :: c \_ d$
(2) $c \rightarrow e :: \_ d$
(3) $\mu \rightarrow \epsilon$

II (1) $\epsilon \rightarrow \mu :: a \_$
(2) $a\mu \rightarrow b :: c \_ d$

5.2. *Mise en œuvre*

Nous avons utilisé nos propres outils à états finis : une bibliothèque de machines à états finis (Beaufort, 2006) définie sur le semi-anneau tropical (Mohri, 2002), et un compilateur de langages rationnels et de règles de réécriture pondérées (Beaufort, 2007), qui construit des machines à états finis au format de notre bibliothèque. Le compilateur autorise l'utilisation des marqueurs. En pratique, un marqueur est déclaré dans une section spéciale, où un identificateur est associé à un index :

```
MARK1    1
```

où `MARK1` est l'identificateur du marqueur, et `1` est son index de référence. Ce marqueur peut ensuite être utilisé dans les règles de réécriture, en utilisant son identificateur entre angles :

```
<MARK1>
```

L'index associé à l'identificateur n'est pas superflu. Dans nos FSM, tout symbole est représenté sous une forme numérique : son numéro d'ordre dans l'alphabet. Le numéro d'ordre que le compilateur attribue à un marqueur μ d'index m sera :

$$\mu = n + m$$

où n correspond au nombre de symboles de l'alphabet utilisé. Définir le numéro d'ordre d'un marqueur en fonction de son index permet donc d'identifier, de manière univoque, un même marqueur dans des ensembles de règles distincts et compilés séparément. Un même marqueur peut de la sorte être inséré, renseigné, identifié et/ou supprimé par différents transducteurs, issus de la compilation de différents ensembles de règles de réécriture.

Afin d'éviter toute ambiguïté, les marqueurs utilisés dans les règles qui illustrent les sections suivantes respectent la syntaxe de notre compilateur. Ils sont écrits en capitales et entourés d'angles. Par exemple, le marqueur *SUB* s'écrira :

```
<SUB>
```

5.3. *Travaux antérieurs*

Dans le cadre des règles de réécriture, le marqueur est classiquement utilisé dans l'algorithme de compilation qui convertit une règle de réécriture en transducteur à état finis. Cet emploi du marqueur a été proposé par Kaplan et Kay (1994) : ils ont mis en évidence que la conversion en transducteur d'une règle de réécriture de la forme $\phi \rightarrow \psi :: \lambda \_ \rho$ nécessite que le compilateur soit capable d'identifier de manière non ambiguë le contexte d'application (λ et ρ) dans lequel s'effectue le remplacement ($\phi \rightarrow \psi$). Dans le cas contraire, le transducteur construit ne correspond pas strictement à la règle. Afin de résoudre ce problème, Kaplan et Kay (1994) ont proposé l'emploi de marqueurs de contexte, utilisés *à l'intérieur du processus de compilation*. De manière succincte, le compilateur décompose une règle de réécriture en une série de transducteurs intermédiaires, et les compose ensemble afin d'obtenir le transducteur désiré. Parmi ces transducteurs intermédiaires, certains insèrent les marqueurs de contexte < et > qui identifient respectivement la fin d'un contexte gauche (<) et le début d'un contexte droit (>), un autre transducteur n'autorise le remplacement $\phi \rightarrow \psi$ que lorsque les marqueurs de contexte sont présents (<ϕ> $\rightarrow$ <ψ>), et un ou plusieurs transducteurs, enfin, se chargent de faire disparaître les marqueurs de contexte du transducteur final. Les marqueurs, dans cet emploi, sont donc internes au compilateur et complètement invisibles pour l'utilisateur qui rédige les règles. Ces marqueurs ne sont donc pas à confondre avec les nôtres, insérés de manière consciente par l'utilisateur afin de s'assurer qu'un phénomène soit correctement géré lorsque plusieurs règles lui sont successivement appliquées.

À la suite de Kaplan et Kay (1994), les marqueurs de contexte ont également été employés dans d'autres compilateurs de règles de réécriture (Karttunen, 1995 ; Karttunen, 1996 ; Mohri et Sproat, 1996 ; Karttunen, 1997), et y remplissent la même fonction. Il faut toutefois noter que chez Karttunen (1995 ; 1996 ; 1997), les marqueurs de contexte sont des *opérateurs*, au même titre que la concaténation ou l'union. Cependant, ce faisant, l'objectif n'était pas de permettre à l'utilisateur de les employer dans la définition de ses règles, mais de faciliter l'implémentation du compilateur. En effet, le compilateur avait été initialement prévu (Karttunen, 1997) pour compiler des règles de réécriture sans contexte. Afin de conserver cette implémentation, Karttunen a préféré décomposer une règle contextualisée en expressions régulières et en règles sans contexte mais pourvues de marqueurs, qui sont converties par le compilateur en transducteurs intermédiaires. Les marqueurs de Karttunen sont donc identiques à ceux de Kaplan et Kay (1994), et ne sont pas à confondre avec les nôtres.

Plus récemment, Kobus *et al.* (2008) ont proposé un système de normalisation des SMS dont certains modules, comme le prétraitement, insèrent des symboles à certains endroits de la chaîne à traiter pour marquer un contexte rencontré, comme le début ou la fin d'une phrase. Dans le principe, ces marqueurs jouent effectivement le même rôle que notre déclencheur. Deux différences distinguent cependant ces marqueurs des nôtres : (1) ce sont toujours des déclencheurs ; (2) ils correspondent à des symboles de l'alphabet, et sont dès lors inutilisables à d'autres fins.

6. Application aux textes dactylographiés

Dans un texte dactylographié, les erreurs résultant en des mots hors vocabulaire (OOV, *out-of-vocabulary word*) sont d'origine typographique ou phonétique (Kukich, 1992). Ce constat était à l'origine du modèle de langue proposé par Toutanova et Moore (2002) (*cf.* section 3.1). Il arrive cependant fréquemment qu'un mot du texte ne diffère de sa référence dans le lexique qu'au niveau de la casse ou de l'accentuation (Yarowsky, 1994), et pourrait être corrigé par un modèle adapté et plus léger.

Le modèle de correction que nous proposons exploite cette idée. Il est construit autour de trois filtres permissifs :
(1) le filtre de casse $Case$, qui modélise les différences de casse et d'accentuation ;
(2) le filtre $Typo$, qui modélise les erreurs typographiques ;
(3) le filtre Pho, qui modélise les erreurs phonétiques.

Le nombre maximal d'erreurs acceptées dans un mot dépend de trois facteurs : la longueur du mot, le type d'erreur et les limites de l'intelligibilité. Nos filtres ont été conçus en gardant ces principes à l'esprit. Dans $Typo$, le nombre d'opérations autorisées dans un mot dépend de sa longueur, et les effets d'une opération donnée dépendent à la fois de son type et de l'agencement du clavier. Dans Pho, les transformations graphémiques dépendent de contraintes intra-lexicales.

6.1. *Le filtre de casse*

Les règles de ce filtre autorisent de simples substitutions 1–1. Par exemple,

[aáàâä]	?→	A	/ 1
[aA]	?→	[áàâä]	/ 1
â	?→	[áàä]	/ 2

Les coûts ont été choisis selon l'hypothèse que les accents sont plus souvent omis que modifiés. Le filtre modélise l'ensemble de l'alphabet latin (ISO-8859-1) sous la forme de 116 règles optionnelles, dont la plupart concernent, comme dans l'exemple précédent, des classes de caractères. Il prend 26 ko sous la forme d'un FST compilé au format binaire de notre bibliothèque.

6.2. *Le filtre typographique*

Nous l'avons mentionné, la longueur du mot, l'opération d'édition et l'agencement du clavier définissent ensemble le nombre, la place et le type d'opérations acceptées. Deux filtres complémentaires modélisent ces contraintes : $Long$, qui fixe les contraintes de longueur, et $Edit$, qui implémente les opérations. $Typo$ est donc obtenu par composition :

$$\mathcal{T}ypo = \mathcal{L}ong \circ \mathcal{E}dit$$

Cependant, $Typo$ n'est jamais calculé. À la place, la liste des candidats est construite au travers de la cascade de transductions :

$$\mathcal{V} = (\,\mathcal{U} \circ \mathcal{L}ong) \circ \mathcal{E}dit\,) \circ \mathcal{L}$$

Les règles de réécriture de $Long$ et $Edit$ utilisent les marqueurs. Dans le principe, les marqueurs sont insérés par $Long$ dans l'OOV en fonction de sa longueur. Ces marqueurs sont des déclencheurs, qui autorisent $Edit$ à appliquer les opérations qu'il décrit. $Long$ est donc un filtre restrictif, qui définit le nombre maximal d'opérations.

Le nombre d'opérations autorisées est défini en fonction d'intervalles de longueur : une opération entre un et cinq caractères, deux entre six et dix caractères, trois entre onze et quinze caractères et quatre lorsque le mot compte plus de quinze caractères. Dans le fichier de règles, les quatre premières règles insèrent des marqueurs qui signalent ces intervalles :

$$\begin{array}{l|l} \epsilon \rightarrow \ \langle \texttt{L1} \rangle \ :: \hat{}\ \ \_\ .\{1,5\} & \epsilon \rightarrow \ \langle \texttt{L3} \rangle \ :: \hat{}\ \ \_\ .\{11,15\} \\ \epsilon \rightarrow \ \langle \texttt{L2} \rangle \ :: \hat{}\ \ \_\ .\{6,10\} & \epsilon \rightarrow \ \langle \texttt{L4} \rangle \ :: \hat{}\ \ \_\ .\{15,\} \end{array}$$

où $\cdot$ signifie « n'importe quel caractère de l'alphabet », $\{n, m\}$ signifie « de n à m occurrences », et $\{n,\}$ signifie « au moins n occurrences ». La règle suivante est optionnelle et insère des marqueurs d'édition après chaque caractère :

$$\epsilon\,? \rightarrow (\langle \texttt{SUB} \rangle | \langle \texttt{INS} \rangle | \langle \texttt{DEL} \rangle | \langle \texttt{TRANS} \rangle) \ :: .\ \_$$

Après application de ces règles sur un FST représentant un mot, certains chemins du FST présentent plus de marqueurs d'édition que la longueur de la séquence ne l'autorise. C'est la raison pour laquelle les règles de réécriture sont suivies d'expressions régulières qui définissent le langage effectivement accepté. Par exemple,

$$\begin{array}{l} [\langle \texttt{L1} \rangle\text{-}\langle \texttt{L4} \rangle]\ .+ \\ [\langle \texttt{L1} \rangle\text{-}\langle \texttt{L4} \rangle]\ .+\ \langle \texttt{ED} \rangle\ .* \\ [\langle \texttt{L2} \rangle\text{-}\langle \texttt{L4} \rangle]\ .+\ \langle \texttt{ED} \rangle\ .+\ \langle \texttt{ED} \rangle\ .* \end{array}$$

où $\langle \texttt{ED} \rangle$ remplace $(\langle \texttt{SUB} \rangle | \langle \texttt{INS} \rangle | \langle \texttt{DEL} \rangle | \langle \texttt{TRANS} \rangle)$ pour des raisons de lisibilité. La première expression n'accepte que les séquences sans marqueurs. La deuxième accepte un seul marqueur, quelle que soit la longueur de la séquence. La troisième accepte deux marqueurs dans les séquences de 6 caractères au moins. Ces expressions, compilées séparément, sont unies dans un seul FSA qui est composé avec le FST représentant les règles de réécriture, ce qui a pour effet de supprimer du FST toutes les séquences qui contenaient plus de marqueurs d'édition que l'intervalle de longueur n'autorisait.
La figure 2b illustre ces règles sur une forme de 5 caractères, marquée par le déclencheur `L1` correspondant. Dans le FST résultant, chaque chemin présente au plus un marqueur d'édition.
Les vraies règles de notre filtre sont un peu plus complexes, mais suivent le même principe. Dans l'ensemble, le langage n'accepte pas d'opérations sur des caractères

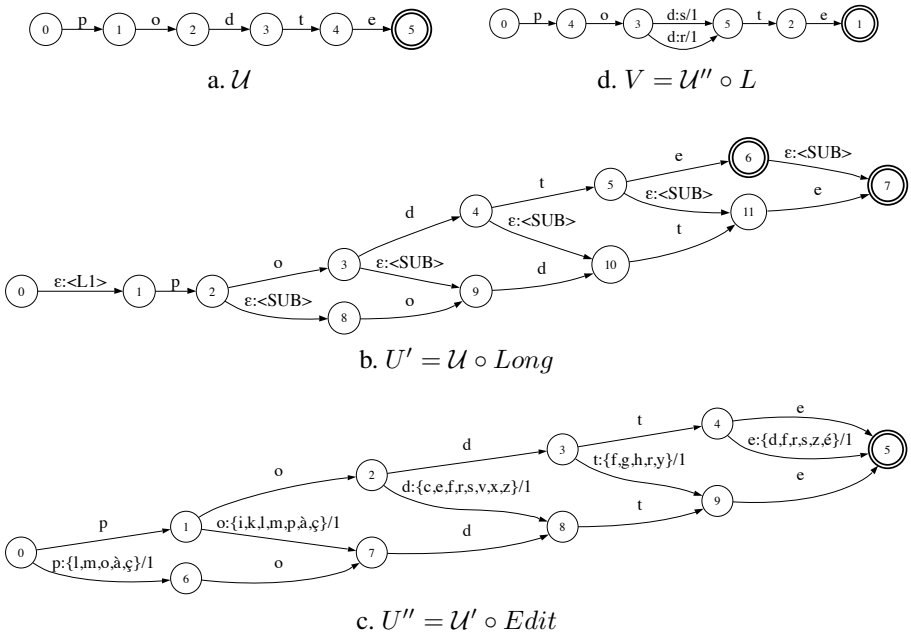


Figure 2. *Application de* Typo *à l'OOV* podte*, une forme corrompue de* porte. *Pour des raisons de lisibilité, nous n'illustrons que les substitutions*

contigus, et refuse que la même séquence présente en même temps insertion, suppression et transposition. Le FST binaire correspondant prend 607 ko.

Les contraintes sur les opérations sont fondées sur les considérations suivantes, illustrées par des règles sans marqueurs :

(1) *A priori*, la substitution dépend fortement de l'agencement du clavier, et devrait fréquemment concerner des caractères proches. De ce fait, nous limitons la substitution aux caractères contigus sur le clavier. Par exemple,

$$d \,?\!\rightarrow [zerfvcxs] \text{ / } 1$$

(2) Supprimer un caractère dépend également de l'agencement du clavier, puisqu'il est probablement plus fréquent d'accrocher un caractère proche qu'un caractère éloigné :

$$d \,?\!\rightarrow \epsilon \;::\; (\_\; [zerfvcxs]) \mid ([zerfvcxs]\; \_) \text{ / } 2$$
$$d \,?\!\rightarrow \epsilon \text{ / } 3$$

Le poids attribué à la suppression de d dépend du contexte. La première règle est favorisée, parce que la suppression est réalisée en présence de caractères appartenant

au voisinage sur le clavier. La troisième règle ne s'applique que dans les autres cas.

(3) Insérer un caractère ne doit pas dépendre du contexte. En effet, le scripteur peut oublier un caractère dans n'importe quel contexte. Une seule règle d'insertion est donc nécessaire :

$\epsilon \stackrel{?}{\rightarrow} .$ / 2

(4) De même, la transposition peut se produire entre n'importe quelle paire de caractères, quelle que soit leur position sur le clavier. Par exemple,

$ab \stackrel{?}{\rightarrow} ba$ / 2,5

Contrairement aux règles illustrant nos contraintes, nos vraies règles exploitent les marqueurs. En voici un exemple, concernant le caractère d et son contexte :

$d\ \langle\texttt{SUB}\rangle\ \rightarrow\ [zerfvcxs]$ / 1
$d\ \langle\texttt{DEL}\rangle\ \rightarrow\ \langle\texttt{ISDEL}\rangle\ ::\ \_\ [zerfvcxs]$ / 2
$d\ \langle\texttt{DEL}\rangle\ \rightarrow\ \langle\texttt{ISDEL}\rangle\ ::\ [zerfvcxs]\ \_$ / 2
$d\ \langle\texttt{DEL}\rangle\ \rightarrow\ \langle\texttt{ISDEL}\rangle$ / 3
$ad\ \langle\texttt{TRANS}\rangle\ \rightarrow\ da$ / 2,5
...
$\langle\texttt{INS}\rangle\ \rightarrow\ .$ / 2
$\langle\texttt{ISDEL}\rangle\ \rightarrow\ \epsilon$

Dans ces règles, le marqueur d'édition est utilisé comme déclencheur : la règle ne peut s'appliquer que si la séquence présente le marqueur approprié. En outre, appliquer la règle supprime le marqueur, de manière à empêcher l'application d'autres règles. On constate enfin que la présence des marqueurs rend les règles obligatoires, contrairement aux règles illustrant nos contraintes, qui étaient optionnelles, mais sans marqueurs. La règle de suppression (marqueur `DEL`) ne supprime pas le symbole, mais le projette sur un autre marqueur, `ISDEL`. Ce marqueur est un bloqueur, qui empêche l'application d'autres règles à la nouvelle séquence que cette suppression aurait créée. Lorsque toutes les règles ont été appliquées, le bloqueur est enfin projeté sur la séquence vide, ϵ, ce qui rend la suppression effective. La figure 2c illustre ces règles. Le filtre contient environ 200 règles et produit un FST de 2,9 Mo.

Cette approche diffère fortement de l'état de l'art, parce que le *nombre* d'opérations autorisées dépend directement de la *longueur* du mot, et que les candidats, non équiprobables, sont directement pondérés par la distance d'édition, en fonction du nombre et du type d'opérations réalisées. Le recours aux marqueurs, en outre, facilite fortement la formulation de contraintes sur les opérations au sein d'un même mot.

6.3. *Le filtre phonétique*

L'objectif de ce filtre est de construire la liste des candidats V en fonction de leur distance phonétique avec l'OOV u. Posons l'OOV *plène*, dont nous supposons qu'il se

prononce [p l ɛ n] ; sur cette base, la distance phonétique devrait construire la liste de candidats {*plaine(s)*, *pleine(s)*}. Or, un système de phonétisation prend normalement en compte la catégorie grammaticale du mot pour décider de sa phonétisation, information qui n'est pas disponible, ou peu fiable, dans le cas d'un OOV. Le système de phonétisation doit donc, dans ce cas, proposer l'ensemble des phonétisations possibles à partir desquelles nous choisirons une liste de candidats dans le lexique phonétisé.

Le filtre phonétique a été construit à partir du lexique phonétisé Brulex (Content *et al.*, 1990), qui compte 282 000 triplets {*forme lexicale*, *catégorie*, *phonétisation*}.

```
couvent          NOUN    k u v a~
couvent          VERB    k u v
```

Les formes lexicales et leurs phonétisations ont été alignées à l'aide de l'algorithme de Black *et al.* (1998). Le lexique aligné se présente comme suit :

```
couvent          NOUN    k u _ v a~ _ _
couvent          VERB    k u _ v _ _ _
```

où les soulignés représentent des silences et facilitent l'alignement graphèmes-phonèmes. Un graphème est « une séquence de caractères prononcée comme un tout ». Dans l'alignement, chaque phonème est aligné sur le premier caractère du graphème correspondant, ce qui facilite la segmentation du mot en sa séquence de graphèmes :

```
c | ou | v | ent        -        c | ou | vent
k   u    v   a~         -        k | u  | v
```

À partir du lexique phonétisé aligné, nous voulons produire des listes de graphèmes homophones, afin de construire des règles de la forme :

$$a \ ?\!\rightarrow \ (b|c|d) \ /w$$

où un graphème a peut être remplacé par les graphèmes homophones b, c ou d, pour un poids w. Ces graphèmes homophones ne se rencontrent cependant pas tous dans les mêmes contextes, et l'idée est d'apprendre ces contextes à l'entraînement, de manière à produire des règles de la forme :

$$a \ ?\!\rightarrow \ b : : \ e \ \_ \ f \ /w_1$$
$$a \ ?\!\rightarrow \ (c|d) : : \ g \ \_ \ h \ /w_2$$

où a ne peut être réécrit b que lorsqu'il est entouré de e et f, parce que la séquence ebf existe dans le lexique. Afin de limiter le nombre de règles du filtre, le contexte, lorsqu'il ne correspond pas à une frontière de mot, a été réduit à un seul caractère. Par exemple *ent* peut être réécrit *ant* en finale de mot après *v*, parce que le lexique contient la forme *devant* :

ent ?→ ant :: v _ $ / 1,5

1 : **Pour tout** mot W du lexique phonétisé,
2 : **Pour tout** graphème Gra de W,
3 : Stocker le triplet (Gra, Pho, G_D) où Pho est le phonème aligné sur Gra, et G_D est le contexte graphémique de Gra
4 : **Pour tout** graphème Gra,
5 : **Pour tout** phonème Pho associé à Gra,
6 : **Pour tout** contexte G_D associé au couple (Gra, Pho)
7 : **Pour tout** graphème Gra_2 prononcé Pho,
8 : Créer une règle "$Gra_2 \,?\!\rightarrow Gra : : G\_D \,/\, 1,5$"

Figure 3. *Algorithme de construction des règles contextuelles du filtre phonétique*

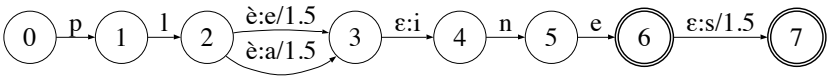


Figure 4. *Application de* Pho *à l'OOV* plène*. Pour plus de lisibilité, la figure ne présente que les réécritures qui appartiennent au lexique utilisé :* pleine *et* plaine *(poids de 1,5), ainsi que leurs pluriels,* pleines *et* plaines *(poids de 3)*

L'algorithme ayant permis d'apprendre automatiquement les règles contextuelles du filtre phonétique est présenté en figure 3. À partir de notre dictionnaire phonétisé, nous avons appris environ 18 000 règles. Le FST binaire correspondant prend 7,3 Mo. L'application de ces règles sur l'OOV *plène* que nous avions posé initialement est présentée en figure 4.

Notre modèle phonétique se distingue de celui de Toutanova et Moore (2002) à deux niveaux. Premièrement, la conversion graphèmes-phonèmes est réalisée lors de l'entraînement, afin de construire un ensemble de graphèmes homophones. Pendant la correction, le FST convertit directement un graphème en ses graphèmes homophones, ce qui supprime l'étape de phonétisation, sans modifier le principe en lui-même. Deuxièmement, la distance phonétique est contextualisée : un graphème n'est utilisé comme remplacement que lorsque le contexte le permet.

6.4. *Évaluation*

Le corpus d'évaluation est fait de 300 phrases regroupant 611 OOV erronés. Il a été construit semi-automatiquement. Une liste de phrases contenant des OOV a d'abord été établie en exécutant notre système d'analyse linguistique (Beaufort et Ruelle, 2006), sans modèle de correction, sur des phrases extraites de journaux et d'e-mails. La liste a ensuite été nettoyée de toutes les formes qui étaient considérées

$Case$	43,04
$Case$ sinon $Typo$	83,80
$Case$ sinon Pho	57,94
$Case$ sinon $(Typo \mid Pho)$	90,18

$Typo = (Case \circ Long \circ Edit)$

Tableau 1. *Textes dactylographiés. Taux de correction (%)*

	1–5 car.	**5–10 car.**	**11–15 car.**	**15+ car.**
$Long \circ Edit$	3,33	10,02	23,23	62,50
$Case \circ (Long \circ Edit)$	6,11	17,43	39,59	92,50
Pho	0,45	0,17	0,45	2,50

Tableau 2. *Textes dactylographiés. Temps de traitement (ms)*

comme correctes : les mots corrects absents à tort du lexique, les noms propres, les mots tronqués, etc. 45 % des OOV retenus présentent au moins deux erreurs d'édition.

Dans les systèmes de l'état de l'art présentés précédemment, l'évaluation des taux de correction se limitait au comptage du nombre de fois où la réponse correcte apparaissait dans les trois meilleures solutions. Dans ce contexte, notre modèle atteint 98,6 %, ce qui est proche des 99 % obtenus par les meilleures approches (Brill et Moore, 2000 ; Toutanova et Moore, 2002), pour autant qu'il soit possible de comparer des méthodes qui ont été évaluées sur des langues et des corpus différents.
Notre évaluation visait cependant surtout à déterminer le comportement du modèle lorsqu'il doit prendre une décision, intégré à notre module d'analyse linguistique automatique. Le tableau 1 présente les configurations testées et les taux de correction obtenus. Quelle que soit la configuration, le système commence toujours par une consultation du dictionnaire au travers du filtre de casse. Les autres filtres ne sont donc utilisés que si cette première consultation échoue. Dans l'ensemble, l'approche semble satisfaisante, puisque le taux de correction tourne autour des 90 % quand tous les modèles de correction sont inclus. On remarque en outre l'importance du filtre de casse, qui assure à lui seul plus de 55 % des corrections et évite donc fréquemment l'emploi de modèles plus coûteux. Le filtre typographique, enfin, corrige plus que le filtre phonétique, mais le cumul de ces deux filtres augmente significativement les résultats.

Les performances de l'approche ont été évaluées sur un PC portable Dell Inspiron 8600 (processeur Intel Pentium Mobile 1,7 GHz, mémoire vive de 1,0 Go) tournant sous Microsoft Windows XP. Le tableau 2 présente les résultats. Dans l'ensemble, nos modèles sont plutôt rapides.
Le filtre phonétique est de loin le moins coûteux. Ceci est dû au fait que ses règles sont

contextualisées : les graphèmes homophones ne peuvent se remplacer l'un l'autre que dans des contextes rencontrés dans le dictionnaire d'entraînement, ce qui limite fortement le nombre des réécritures.
Le filtre de casse est en revanche le plus gourmand, ce qui s'explique par le fait que les remplacements qu'il modélise (changement d'accentuation, changement de casse) peuvent être réalisés n'importe où, indépendamment du contexte. Cette gourmandise reste toutefois dans les limites du raisonnable, si l'on tient compte du point auquel ce filtre contribue au taux de correction.
Il est par ailleurs intéressant de souligner que la gourmandise d'un FST n'est en rien proportionnelle à sa taille ni au nombre de ses règles. Comme nous avons eu l'occasion de le mentionner, le filtre de casse ne prend que 26 ko et modélise à peine 116 règles, alors que le filtre phonétique prend 7,3 Mo et correspond à environ 18 000 règles. Les règles du filtre de casse, en revanche, sont non contextualisées, au contraire de celles du filtre phonétique. Nous en concluons que le potentiel génératif d'un FST dépend à la fois du nombre de remplacements qu'il autorise et des contraintes contextuelles exprimées sur ces remplacements.

7. Application à la reconnaissance optique des caractères

La démocratisation des prix des appareils photos numériques de toutes qualités, depuis le matériel d'entrée de gamme jusqu'au matériel professionnel, a rendu la prise de photos numériques à la portée de tous. Ceci a ouvert la voie au développement de nombreuses applications visant à extraire – voire à *comprendre* – le texte présent dans ces photos. Parmi ces applications, figurent en très bonne place les systèmes d'aide aux personnes visuellement handicapées, qui reposent sur des systèmes de reconnaissance des caractères (OCR, *optical character recognition*). C'est le cas de l'application qui accueille le modèle de correction présenté dans cette section : un assistant mobile de lecture pour personnes aveugles et malvoyantes (Mancas-Thillou, 2006). Installée sur un PDA[3], cette application permet à l'utilisateur de prendre des photos, dont les zones de texte sont extraites, reconnues par un OCR et lues automatiquement par un système de synthèse de la parole.

De nombreux facteurs rendent cependant difficile la reconnaissance des textes présents dans ces photos. La photo peut, en effet, présenter de nombreuses dégradations, du fait de problèmes de luminosité, d'exposition ou de focus, ou à cause de la résolution choisie ou de capteurs bruités qui, par exemple, rendent les couleurs imprécises. Le cadrage peut en outre être à l'origine de caractères coupés ou de mots tronqués. Enfin, l'orientation du texte peut être en cause, mais également la variété des polices de caractères utilisées, dont certaines présentent des effets artistiques prononcés. L'ensemble de ces caractéristiques, que l'on retrouve rarement dans les documents numérisés à l'aide de scanners à plat, classent les photos prises par les appareils numériques dans la catégorie des *scènes naturelles*.

3. PDA : *Personal Digital Assistant*, assistant personnel.

Définition 3 (scène naturelle) *En traitement d'images, une* scène naturelle *est une image photo ou une trame de film vidéo prise dans le monde réel, sans aucune contrainte.*

Confronté à une scène naturelle, l'OCR n'a aucune connaissance *a priori*, ni aucune information fiable concernant l'environnement, les conditions d'éclairage, ni la disposition des objets ou du texte dans l'image. Un OCR, même lorsqu'il est dédié aux scènes naturelles, est donc plus enclin aux erreurs que les OCR utilisés dans les scanners à plat. Une étape de correction après l'étape de reconnaissance est dès lors nécessaire.

7.1. *Principes du modèle de correction*

Un OCR comporte classiquement deux modules. Premièrement, le module de segmentation, qui détecte le texte dans l'image, l'extrait et le coupe en segments. Deuxièmement, le classificateur, qui attribue une classe – un caractère – à chaque segment. Dans ce système, les erreurs ont trois causes. Premièrement, le module de segmentation, qui propose de mauvais segments. Dans ce cas, le classificateur est incapable d'attribuer une étiquette correcte au caractère, et nous sommes en présence de confusions n–m, telles que {*m*, *rn*} ou {*li*, *h*}. Deuxièmement, le classificateur, qui distingue difficilement certaines paires de caractères, telles que {*i*, *l*}. Il s'agit de confusions 1–1. Troisièmement, le nombre de classes du classificateur. En scènes naturelles, 36 classes sont généralement prises en compte : les 26 minuscules de l'alphabet latin et les dix chiffres arabes de 0 à 9. Majuscules et lettres accentuées ne sont pas modélisées par le classificateur, mais confondues avec la minuscule correspondante, parce que les bases d'entraînement disponibles ne contiennent pas assez d'exemplaires de majuscules et que les accents ont tendance à se confondre avec le bruit des images à traiter (Mancas-Thillou, 2006). C'est ce que nous appelons la *limite de casse*. Bien sûr, une erreur peut présenter à la fois une confusion et une limite de casse. Par exemple, *é* pourrait être classé *c* au lieu de *e*.

Le modèle de correction que nous proposons se caractérise comme suit :

(1) il tire avantage des taux de confiance fournis par le classificateur : pour chaque caractère, les trois meilleures classes de l'OCR sont conservées. La forme lexicale u est donc remplacée ici par un treillis de formes lexicales, le WFSA $\mathcal{W}$;

(2) le cœur du modèle est un filtre adapté aux erreurs spécifiques de l'OCR. Appelé $Reco$, il se place entre le treillis et le lexique :

$$\mathcal{V} = \mathcal{W} \circ \mathcal{R}eco \circ \mathcal{L}$$

(3) afin de gérer les erreurs présentant simultanément confusion et limite de casse, $Reco$ est en fait le résultat de la composition suivante :

$$\mathcal{R}eco = \mathcal{C}onfus \circ \mathcal{C}ase_2$$

où $Confus$ gère à la fois les confusions 1–1 et n–m, et $Case_2$ s'occupe de la limite de casse ;

(4) le modèle complet a été pensé pour les plates-formes embarquées. Nous avons tâché de réduire les besoin en mémoire vive. Dans ce but, les deux parties du filtre sont conservées séparées, et la consultation du lexique est réalisée au travers de la cascade de compositions suivante :

$$\mathcal{V} = (\,(\mathcal{W} \circ \mathcal{C}onfus) \circ \mathcal{C}ase_2\,) \circ \mathcal{L}$$

Dans cette cascade, le FST obtenu après chaque composition est optimisé : nous le réduisons à l'automate correspondant à sa seconde projection[4], et lui appliquons, si nécessaire, la suppression-ϵ et la déterminisation.

7.2. *Le filtre de confusion*

Le modèle de confusion a été entraîné sur les erreurs de l'OCR, exécuté sur les images du corpus de scènes naturelles « ICDAR 2003 » (Lucas *et al.*, 2003). Nous avons comparé l'étiquetage du corpus à l'étiquetage produit par l'OCR, et mémorisé dans une liste les erreurs de l'OCR et leur nombre d'occurrences. Afin d'obtenir un modèle indépendant du contexte, nous avons suivi deux principes. Premièrement, une erreur de l'OCR ne peut inclure de caractère correct. Deuxièmement, nous avons rejeté les confusions n–m ($n, m > 1$) au profit des confusions 1–m et n–1, qui correspondent mieux aux erreurs du module de segmentation. Ces deux principes distinguent notre modèle de celui de Kolak *et al.* (2003). Enfin, les erreurs qui ne s'étaient produites qu'une seule fois ont été supprimées de la liste, limitée de ce fait à 221 confusions. Les probabilités associées à ces confusions correspondent à :

$$P(b|a) = \frac{\#(a \mapsto b)}{\#(a)} \qquad [19]$$

où a est l'erreur, b est la correction et $(a \mapsto b)$ signifie « a doit être réécrit b ». Toutes les probabilités ont donc été calculées de la même façon, que la confusion concerne un seul ou plusieurs caractères. Ceci permet de tenir compte du fait que la seule information connue est la séquence a produite par l'OCR.

Le nombre d'erreurs autorisées est limité à un tiers de la longueur du mot. Ceci n'entraîne aucune perte de qualité : sur des scènes naturelles, le taux de reconnaissance de l'OCR dépasse les 80 % (Thillou *et al.*, 2005). Le nombre d'erreurs aurait donc pu être limité à un cinquième de la longueur.

4. Posons un transducteur T. Par définition, chaque transition de T est de la forme (q, a_1, a_2, r), où q est l'état de départ et r est l'état d'arrivée de la transition, a_1 est le premier symbole de la transition et est défini sur un alphabet Σ_1, et a_2 est le second symbole de la transition et est défini sur un alphabet Σ_2. Réduire T à sa première ou à sa seconde projection revient à le réduire à un automate, en ne conservant de chaque transition que le premier symbole (q, a_1, r) (première projection) ou le second (q, a_2, r) (seconde projection).

Les contraintes de longueur et les confusions sont modélisées séparément. Les règles de longueur marquent des intervalles de longueur à l'aide de marqueurs dédiés (de `L1` à `L4`) :

$$\epsilon \rightarrow \langle \texttt{L1} \rangle \ :: \ \hat{} \ \_ \ .\{1,3\}$$
$$\ldots$$
$$\epsilon \rightarrow \langle \texttt{L4} \rangle \ :: \ \hat{} \ \_ \ .\{10,\}$$

Chaque règle de confusion corrige une confusion donnée et ajoute un marqueur (de `C1` à `C4`) qui indique le nombre de confusions qui ont déjà été appliquées à la séquence en cours. Une règle ne peut s'appliquer que si le marqueur de longueur adéquat ainsi que tous les marqueurs de confusion inférieurs au sien sont présents dans la séquence :

$$0 \ ? \rightarrow 8 \ \langle \texttt{C1} \rangle \ :: \ [\langle \text{L1} \rangle \langle \text{L2} \rangle \langle \text{L3} \rangle \langle \text{L4} \rangle] \ .* \ \_ \ .* \$ \ / \ 6,8579$$
$$m \ ? \rightarrow rn \ \langle \texttt{C1} \rangle \ :: \ [\langle \text{L1} \rangle \langle \text{L2} \rangle \langle \text{L3} \rangle \langle \text{L4} \rangle] \ .* \ \_ \ .* \$ \ / \ 4,8579$$
$$\ldots$$
$$0 \ ? \rightarrow 8 \ \langle \texttt{C2} \rangle \ :: \ [\langle \text{L2} \rangle \langle \text{L3} \rangle \langle \text{L4} \rangle] \ .+ \ \langle \text{C1} \rangle \ .* \ \_ \ .* \$ \ / \ 6,8579$$
$$m \ ? \rightarrow rn \ \langle \texttt{C2} \rangle \ :: \ [\langle \text{L2} \rangle \langle \text{L3} \rangle \langle \text{L4} \rangle] \ .+ \ \langle \text{C1} \rangle \ .* \ \_ \ .* \$ \ / \ 4,8579$$

Une confusion de niveau n ne peut donc s'appliquer que si toutes les confusions de 1 à $n-1$ ont déjà été appliquées. La dernière règle supprime enfin tous les marqueurs :

$$[\langle \text{L1} \rangle \text{-} \langle \text{L4} \rangle \langle \text{C1} \rangle \text{-} \langle \text{C4} \rangle] \ \rightarrow \ \epsilon$$

Compilé sous la forme d'un FST binaire, le filtre prend 2,2 Mo.

7.3. *Le filtre de casse*

Le filtre de casse appliqué à la sortie de l'OCR est un sous-ensemble de celui conçu pour les textes dactylographiés. Deux raisons justifient ce choix. Premièrement, seuls les caractères minuscules non accentués doivent être modélisés :

a ?→ [Aäâáàã]
n ?→ [Nñ]

Deuxièmement, les règles sont optionnelles mais *non pondérées*, comme on le voit dans l'exemple précédent. Ceci est dû au fait que, quelles que soient sa casse et son accentuation, tout caractère est systématiquement converti en la minuscule non accentuée correspondante. De ce fait, utiliser une distance d'édition désavantagerait injustement les majuscules et les lettres accentuées. Le FST correspondant prend 1,5 ko.

Cela dit, le filtre de casse pourrait être pondéré. Dans ce cas, le meilleur choix serait certainement un modèle de langue estimant la probabilité qu'une forme donnée a_i appartiennent à un ensemble $\overline{a}$ de formes ne différant que par la casse et/ou l'accentuation :

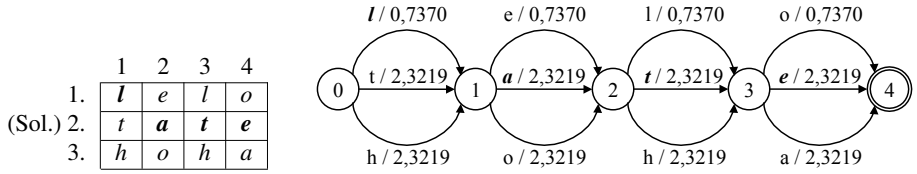
$$\forall a_i \in \overline{a}, \ P(a_i|\overline{a}) = \frac{\#(a_i, \overline{a})}{\#(\overline{a})} \qquad [20]$$

Dans ce modèle, la minuscule non accentuée serait pondérée également. Un tel modèle, malheureusement, serait dépendant de la langue…

7.4. *Le treillis lexical*

En intégrant les trois meilleures sorties de l'OCR pour chaque caractère, le WFSA W est un moyen simple d'autoriser toutes les successions de caractères. Il rend l'algorithme plus facile, étant donné que toutes les possibilités ne doivent pas être testées séquentiellement. La figure 5 en donne un exemple. Dans W, chaque sortie de l'OCR reçoit un poids. Notre hypothèse est que la meilleure sortie de l'OCR, qui totalise généralement plus de 90 % du taux de confiance, doit être privilégiée au détriment des deux autres, que l'OCR lui-même distingue difficilement. Sur cette base, nous avons attribué à la première sortie un poids trois fois plus important (0,6) qu'aux deux autres (0,2). La bibliothèque de FSM étant définie sur le semi-anneau tropical $(\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0)$, les poids sont représentés dans l'automate sous la forme de logarithmes négatifs en base 2 :

$$\begin{array}{l} -\log_2(0,6) = 0,7370 \\ -\log_2(0,2) = 2,3219 \end{array} \qquad [21]$$



Les trois meilleures classes — $\mathcal{W}$, le WFSA correspondant

Figure 5. *Construction du treillis lexical $\mathcal{W}$. Le mot à reconnaître est* late *(« tard »)*

7.5. *Évaluation*

Nous rappelons que la liste de confusion à la base du filtre de confusion a été apprise sur le corpus anglais constitué par ICDAR 2003 (Lucas *et al.*, 2003).

Les taux de correction ont été évalués en français et en anglais. En français, l'évaluation a été réalisée sur un corpus de notre composition, contenant 400 scènes naturelles. Nous la comparons au taux de correction obtenu sur le même corpus par un simple trigramme de lettres : 86,5 % (Thillou *et al.*, 2005). La composition filtrée, elle, monte à 94,7 %. Ce bon résultat sur un corpus français tend à prouver que la liste de confusion n'est pas spécifique à la langue du corpus sur lequel elle a été apprise, mais modélise bien les erreurs du système lui-même.

	Taux de correction (%)	**Bruit** (%)
ABBYY (3 meilleures suggestions)	34,0	0,0
Composition filtrée (sans treillis)	54,3	0,6
Composition filtrée (avec treillis)	65,4	8,6

Tableau 3. *Scènes naturelles. Évaluation sur le corpus ICDAR 2003*

(a)  (b) 

Figure 6. *Deux scènes naturelles difficiles*

En anglais, l'évaluation a porté sur les 2 268 scènes naturelles du corpus ICDAR 2003 lui-même. Elle compare notre approche au système commercial *ABBYY FineReader 8.0 Professional Edition Try&Buy*. Après une segmentation des images adaptée aux scènes naturelles (Mancas-Thillou, 2006), les segments ont été traités par l'OCR commercial, avec un taux d'erreurs de 29,1 %. L'évaluation n'a porté que sur ces erreurs. Le tableau 3 compare les trois premières suggestions de l'OCR commercial à la correction automatique produite par la composition filtrée, appliquée soit au treillis lexical, soit à la meilleure sortie de l'OCR uniquement. Notre système surpasse indéniablement le système commercial, et le treillis lexical améliore considérablement les résultats. Le seul inconvénient à l'emploi de ce treillis est la confusion qu'il introduit, puisque le nombre d'OOV modifiés à tort (bruit) augmente significativement.
Notre correcteur gère efficacement les substitutions n–m. Par exemple, il corrige le mot *Office* (figure 6a), reconnu *Ohice* par l'OCR commercial. Le système s'en sort également avec les caractères coupés, si typiques des scènes naturelles, comme dans le mot *form* (figure 6b). Sans correction, notre OCR avait reconnu *fnrm*. Grâce au treillis lexical où *o* est la deuxième sortie pour le deuxième caractère, la correction, elle, arrive à classer *form* devant *farm* et à proposer la bonne solution.

Les performances du système ont été évaluées sur un PDA classique tournant sous Microsoft Pocket PC 2003 et pourvu d'un processeur Intel 520 MHz et de 64 Mo de SDRAM[5]. Le corpus d'évaluation était notre corpus français de 400 scènes naturelles. En moyenne, les mots comptant jusqu'à 13 caractères sont corrigés en 0,2 s. Cette performance, de même que la possibilité d'utiliser le système sur un PDA, sont des conséquences directes des étapes de simplification réalisées après chaque composition de la cascade présentée. Sans ces simplifications, le système dépasserait tout simplement la place mémoire disponible.

5. SDRAM : *Synchronous Dynamic Random Access Memory*.

8. Conclusions et perspectives

La distance d'édition est un bon moyen de modéliser de manière explicite les erreurs commises par le média d'acquisition du texte, qu'il s'agisse d'un être humain ou d'un système OCR. Cependant, l'implémentation classique de cette méthode comporte deux inconvénients majeurs, au centre de nombreuses études. Premièrement, la pondération n'est pas efficace, et les études focalisées sur ce point l'ont remplacée par des modèles de langue. Parmi celles-ci, la seule approche capable de gérer efficacement les erreurs n–m est celle de Toutanova et Moore (2002), mais au prix d'une étape supplémentaire de phonétisation. Deuxièmement, l'algorithme de recherche ne traite qu'une forme du lexique à la fois, ce qui est beaucoup trop coûteux en présence de lexiques de taille réelle. Les études centrées sur cette question ont utilisé des machines à états finis. Parmi ces approches, seule celle de Mohri (2003) propose un modèle de pondération efficace, modèle sur lequel se fonde l'approche que nous avons présentée.

Notre méthode, également fondée sur des machines à états finis, est nettement distincte des méthodes précédentes :

(1) la liste des candidats V est le résultat de la *composition filtrée* $\mathcal{U} \circ \mathcal{F} \circ \mathcal{L}$, où $\mathcal{F}$, qui modélise les opérations d'édition valides, est le *filtre* qui fixe la taille de l'intersection entre $\mathcal{U}$ et $\mathcal{L}$;

(2) dans ce modèle, toutes les données sont représentées sous la forme de machines à états finis. L'approche dans son ensemble offre de ce fait une grande flexibilité : d'une part, de l'information externe au module de correction peut facilement être intégrée au processus de pondération, et d'autre part, la liste des candidats est un FSM qui peut subir d'autres « analyses à états finis » avant qu'une décision ne soit prise ;

(3) le filtre $\mathcal{F}$ est toujours un WFST obtenu par compilation de règles de réécriture, soit conçues par un expert, comme le filtre de casse, soit apprises et pondérées sur un corpus, comme le filtre de confusion de l'OCR ;

(4) le filtre $\mathcal{F}$ peut être le résultat d'une cascade de compositions, ce qui facilite la modélisation d'opérations complexes à partir d'opérations simples ;

(5) conserver $\mathcal{F}$ sous la forme d'une cascade de compositions au lieu de le compiler sous la forme d'un seul FST réduit les besoins en temps et en espace, ce qui permet au système de tourner sur plate-forme embarquée ;

(6) les marqueurs de règles facilitent l'expression de conditions et de contraintes. À l'aide de marqueurs, par exemple, il est facile de détecter la *longueur* d'un mot dans une règle et d'en tenir compte dans une autre pour déterminer le *nombre maximal* d'opérations d'édition.

Dans cet article, par souci de clarté, nous avons volontairement passé certains faits sous silence. D'une part, le texte d'une scène naturelle est généralement dactylographié. Il peut donc contenir des erreurs dues au scripteur. À l'aide des modèles présentés, gérer à la fois les erreurs de l'OCR et du scripteur est assez

simple, si l'on tient compte du fait que les sources des erreurs sont séquentielles, avec dans l'ordre le scripteur, puis l'OCR. Dans cette perspective, la consultation du dictionnaire se réalise au travers de la cascade de compositions suivante :

$$\mathcal{V} = \mathcal{W} \circ \mathcal{R}eco \circ \mathcal{C}ase \circ \mathcal{L}$$

où $\mathcal{W}$ est le treillis de solutions lexicales construit à partir de la sortie de l'OCR. Si cette consultation échoue, l'étape réelle de correction correspond dans ce cas à la cascade de compositions suivante :

$$\mathcal{V} = \mathcal{W} \circ \mathcal{R}eco \circ (\mathcal{T}ypo \,|\, \mathcal{P}honet) \circ \mathcal{L}$$

D'autre part, les applications présentées dans cet article concernent uniquement les OOV. Les erreurs contextuelles, cependant, peuvent également profiter de cette approche. Les flexions homophones d'un lemme donné, par exemple, peuvent être considérées comme des confusions phonétiques n–m et être modélisées comme suit :

ées ?→ é | ée | és | er | ez | ai

Dans ce filtre, les variantes homophones peuvent soit recevoir le même poids,

ées ?→ (é | ée | és | er | ez | ai) / 1

soit bénéficier d'une expertise linguistique, afin de distinguer ce qui peut l'être :

1. ées ?→ (é | ée | és) / 0,5
2. ées ?→ (er | ez) / 0,75
3. ées ?→ ai / 1
4. ées ?→ ais / 2

Ce dernier exemple montre que la composition filtrée est avant tout un *principe*, qui permet de décrire de manière flexible la taille de l'intersection entre deux langages.

Remerciements

Je remercie sincèrement Sophie Roekhaut, Alain Ruelle et Céline Thillou, pour les conseils qu'ils m'ont donnés et les discussions stimulantes que nous avons eues. Je remercie également tous mes relecteurs, dont Anne-Sophie Dassy et Thomas François, pour leurs corrections et suggestions sur les versions antérieures de cet article.

9. Bibliographie

Aho A., « Algorithms for finding patterns in strings », *Handbook of Theoretical Computer Science*, Elseviers Science Publishers, B.V., Amsterdam, p. 256-300, 1990.

Aho A., Sethi R., Ullman J., *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

Beaufort R., FSM library : Description de l'API, Technical report, Multitel ASBL, `http://cental.fltr.ucl.ac.be/team/beaufort/publi.html`, 2006.

Beaufort R., Ovide documentation. From regular expressions and rewrite rules to finite-state machines, Technical report, Multitel ASBL, `http://cental.fltr.ucl.ac.be/team/beaufort/publi.html`, 2007. Version 1.4.2.

Beaufort R., Dutoit T., Pagel V., « Analyse syntaxique du français. Pondération par trigrammes lissés et classes d'ambiguïtés lexicales », *Proc. JEP*, p. 133-136, 2002.

Beaufort R., Ruelle A., « eLite : Système de synthèse de la parole à orientation linguistique », *Proc. JEP*, p. 509-512, 2006.

Black A., Lenzo K., Pagel V., « Issues in building general letter-to-sound rules », *Proc. 3rd ESCA/COCSADA Workshop on Speech Synthesis*, p. 77-81, 1998.

Brill E., Moore R. C., « An improved error model for noisy channel spelling correction », *Proc. ACL 2000*, p. 286-293, 2000.

Chomsky N., « Three models for the description of language », *I.R.E. Transactions on Information Theory*, vol. IT-2, n° 3, p. 113-124, 1956.

Church K., Gale W., « Probability Scoring for Spelling Correction », *Statistics and Computing*, vol. 1, p. 93-103, 1991.

Content A., Mousty P., Radeau M., « Brulex, une base de donnée lexicales informatisée pour le français écrit et parlé », *L'année Psychologique*, vol. 90, p. 551-566, 1990.

Damerau F., « A technique for computer detection and correction of spelling errors », *Communications of the ACM*, vol. 7, n° 3, p. 171-176, 1964.

Eilenberg S., *Automata, Languages, and Machines*, Academic Press Inc., Orlando, FL, USA, 1974.

Fisher W., « A statistical text-to-phone function using ngrams and rules », *Proc. of ICASSP*, vol. 2, p. 649-652, 1999.

Hopcroft J., Motwani R., Ullman J. (eds), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Massachusetts, 1979.

Johnson C. D., *Formal Aspects of Phonological Description*, Mouton, The Hague, 1972.

Kaplan R., Kay M., « Regular models of phonological rule systems », *Computational Linguistics*, vol. 20, n° 3, p. 331-378, 1994.

Karttunen L., « The replace operator », *Proc. ACL'95*, p. 16-23, 1995.

Karttunen L., « Directed replacement », *Proc. ACL'96*, p. 108-115, 1996.

Karttunen L., « The replace operator », *in* E. Roche, Y. Schabes (eds), *Finite-State Language Processing, chapter 4*, MIT Press, p. 117-147, 1997.

Kleene S., « Representation of events in nerve nets and finite automata », *Automata Studies*, 1956.

Kobus C., Yvon F., Damnati G., « Transcrire les SMS comme on reconnaît la parole », *Actes de la Conférence sur le Traitement Automatique des Langues (TALN'08)*, Avignon, France, p. 128-138, 2008.

Kolak O., Byrne W., Resnik P., « A generative probabilistic OCR model for NLP applications », *Proc. NAACL-Human Language Technology*, vol. 1, p. 55-62, 2003.

Kuich W., Salomaa A., *Semirings, Automata, Languages*, vol. 5 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin, Germany, 1986.

Kukich K., « Techniques for Automatically Correcting Words in Text », *ACM Computing Surveys*, vol. 24(4), p. 377-439, 1992.

Levenshtein V., « Binary codes capable of correcting deletions, insertions and reversals », *Soviet Physics*, vol. 10, p. 707-710, 1966.

Lucas S., Panaretos A., Sosa L., Tang A., Wong S., Young R., *Robust Reading Competition*, Washington, DC, USA. 2003.

Mancas-Thillou C., Natural Scene Text Understanding, PhD thesis, FPMS, Mons, Belgium, 2006.

McNaughton R., Yamada H., « Regular expressions and state graphs for automata », *IRE Transactions on Electronic Computers EC*, vol. 9, n° 1, p. 39-47, March, 1960.

Miki B., McHugh S., « Selectable marker genes in transgenic plants : Applications, alternatives and biosafety », *Journal of Biotechnology*, vol. 107, n° 3, p. 193-232, February, 2004.

Mohri M., « On some applications of finite-state automata theory to natural language processing », *Journal of Natural Language Engineering*, vol. 2, p. 1-20, 1996.

Mohri M., « Semiring frameworks and algorithms for shortest-distance problems », *Journal of Automata, Languages and Combinatorics*, vol. 7, n° 3, p. 321-350, 2002.

Mohri M., « Edit-distance of weighted automata », *Lecture Notes in Computer Science*, vol. 2608, p. 1-23, 2003.

Mohri M., Pereira F., Riley M., « The design principles of a weighted finite-state transducer library », *Theoretical Computer Science*, vol. 231, n° 1, p. 17-32, 2000.

Mohri M., Sproat R., « An Efficient Compiler for Weighted Rewrite Rules », *Proc. ACL'96*, p. 231-238, 1996.

Oflazer K., « Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction », *Computational Linguistics*, vol. 22, n° 1, p. 73-89, 1996.

Pereira F., Riley M., Sproat R., « Weighted rational transductions and their application to human language processing », *Human Language Technology Workshop*, p. 262-267, 1994.

Peterson J., « A note on undetected typing errors », *Communications of the ACM*, vol. 29(7), p. 633-637, 1986.

Schulz K., Mihov S., « Fast string correction with Levenshtein-automata », *International Journal of Document Analysis and Recognition*, vol. 5, n° 1, p. 67-85, 2002.

Schützenberger M., « On the definition of a family of automata », *Information and Control*, vol. 4, p. 245-270, 1961.

Thillou C., Ferreira S., Gosselin B., « An embedded application for degraded text recognition », *Eurasip Jour. on Applied Signal Processing*, vol. 13, p. 2127-2135, 2005.

Toutanova K., Moore R. C., « Pronunciation modeling for improved spelling correction », *Proc. ACL'02*, p. 144-151, 2002.

Wagner R., « Order-n correction for regular languages », *Communications of the ACM*, vol. 17, n° 5, p. 265-268, 1974.

Yannakoudakis E., Fawthrop D., « The rules of spelling errors », *Information Processing and Managment*, vol. 19, n° 2, p. 87-99, 1983.

Yarowsky D., « Decision lists for lexical ambiguity resolution : application to accent restoration in Spanish and French », *Proc. ACL'94*, p. 88-95, 1994.