TP OCaml n°3

Spellchecker

I. Introduction.

Le but de ce TP est de réaliser un *"spellchecker"* ou « vérificateur orthographique » en français. Pour rappel, dans un outil d'édition de document tel que Libre Office Write, l'écriture d'un mot qui n'est pas dans la langue française entraine l'avertissement que ce mot n'est pas reconnu, et de comment le corriger.

Dans ce TP, on ne s'intéressera qu'à la correction orthographique, pas à la correction grammaticale. Une succession de mots valides sera considérée valide même si elle ne respecte pas les règles de la grammaire française (sujet + verbe + complément). De plus, afin d'éviter des problèmes liés aux accents,[1] nous utiliserons la langue anglaise.

Un peu d'histoire. Le premier *spellchecker*, nommé SPELL, date de 1971. Son créateur, Ralph Gorin, a inclus une liste de 10 000 mots anglais dans son programme Assembly. Les algorithmes et structures de données utilisées par ce programme inspirera l'outil *ispell* d'Unix. Ce programme pouvait, en effet, *détecter* les erreurs mais aussi *suggérer* des modifications.

Dans les fichiers associés à ce TP, vous trouverez `dictionary.txt` qui contient, une liste des 10 000 mots anglais les plus utilisés.

II. Deteksion de fautes d'ortografe.

Dans cette partie, on nous donne un texte, une suite de mots, et on se charge de détecter les fautes d'orthographe. C'est la « partie simple » du *spellchecker*.

Q1. En OCaml, définir une variable `dico: string list` qui correspond à la liste des mots du fichier `dictionary.txt`.

Des structures de données bien plus efficaces existent pour stocker une liste de mots. On peut citer, par exemple, les arbres préfixes.[2]

Q2. Écrire une fonction de signature `split_on_char : char -> string -> string list` qui, lors de l'appel `split_on_char x s` découpe la chaîne `s` en différentes parties séparées par `x`. On n'utilisera pas la fonction `String.split_on_char`.

Par exemple, l'appel `split_on_char 'x' "aaxaabxaaxaxxa"` découpera la chaîne donnée en les sous-chaînes `["aa";"aab";"aa";"a";"";"a"]`.

Q3. Écrire une fonction de signature `spellcheck : string -> string list` de telle sorte que la fonction renvoie, pour un texte donné, les mots qui ne sont pas correctement orthographiés. On utilisera *habilement* les questions précédentes. On ne se souciera pas de la ponctuation.

Q4. Estimer la complexité de la fonction codée en Q3 en fonction de m la taille du dictionnaire en nombre de mots, et n la taille du texte en nombre de mots.

Une bonne structure de données permet d'améliorer la complexité trouvée en Q4.

[1] Les lettres accentuées telle que « é » ne font pas parties de la table ASCII, mais du standard Unicode, le traitement est donc plus compliqué en C comme en OCaml.

[2] Il est probable que vous l'ayez en exercice de préparation aux oraux.

III. Correction orthographique.

Pour le moment, nous avons codés un *vérificateur* orthographique, il est temps d'ajouter un *correcteur* orthographique. Ce que l'on souhaite, c'est que, lorsqu'on écrit le mot « chajeau », on nous propose différentes variantes valides « chapeau », « chateau », « chameau », *etc.*

La difficulté est : quels mots choisir ? Il faut définir une notion de resemblance, de distance, entre deux mots.

III.1. Distance d'édition.

Définition. Soient u et v deux mots d'un alphabet Σ. On définit la *distance de Levenshtein*, que l'on notera $\mathrm{lev}(u, v)$, comme le nombre minimum de les seules opérations élémentaires au niveau d'un caractère (insertions, suppressions ou substitutions) nécessaires pour changer un mot en un autre. On définit donc récursivement :

$$\mathrm{lev}(u,v) = \begin{cases} |u| + |v| & \text{si } |u| = 0 \text{ ou } |v| = 0 \\ \mathrm{lev}(\mathrm{tail}(u), \mathrm{tail}(v)) & \text{si } \mathrm{head}(u) = \mathrm{head}(v) \\ 1 + \min \begin{pmatrix} \mathrm{lev}(\mathrm{tail}(u), v), \\ \mathrm{lev}(u, \mathrm{tail}(v)), \\ \mathrm{lev}(\mathrm{tail}(u), \mathrm{tail}(v)) \end{pmatrix} & \text{sinon} \end{cases}$$

où $\mathrm{head}(x) = x_1$, la première lettre de x, et $\mathrm{tail}(x) = x_2 \dots x_n$, le mot x sans la première lettre.

La distance de Levenshtein est également nommée *distance d'édition*.

Dans la suite, la *distance* considérée sera toujours la distance de Levenshtein.

Q5. Prouver que $\mathrm{lev}(u, u) = 0$ pour $u \in \Sigma^\star$. On pourra procéder par récurrence sur $|u|$.

Q6. Coder la fonction `lev_distance : string -> string -> int` qui calcule $\mathrm{lev}($`u`$,$ `v` $)$ lors de l'appel `lev_distance u v`.[3] Quelle est la complexité de la fonction codée ?

Q7. *À la main*, quelle est la distance entre les mots « chien » et « chat » ? On dessinera un arbre avec les « appels récursifs » de la fonction lev.
Que remarquez vous ? Est-il possible d'améliorer la complexité obtenue en Q6 ?

Q8. Implémenter l'amélioration trouvée en Q7. Votre solution devrait avoir une complexité de l'ordre de $\Theta(|u| \cdot |v|)$ au moins.[4]

Q9. À l'aide d'un invariant bien choisi, montrer que la fonction calcule toujours la distance d'édition.

III.2. Suggestions de mots.

Q10. En utilisant toutes les questions précédentes, coder une fonction `suggest` ayant pour signature `suggest : int -> string -> (string * int) list` telle que `suggest` $k\,u$ retourne une liste de k couples (v, d) où $d = \mathrm{lev}(u, v)$ et les mots v sont les k premiers mots du dictionnaire par rapport à la distance d'édition à u. On prendra soin de ne pas réaliser d'opérations trop coûteuses.

Q11. Quelle est la complexité de la solution Q10 ? Vous devriez avoir une complexité en $\Theta(nk)$ où n est la taille du dictionnaire et k est donné en paramètre.

[3] On implémentera *exactement* la fonction comme elle est décrite dans sa version mathématique, sans optimisation supplémentaire.

[4] Il est possible d'obtenir une complexité en $\Theta(\max(|u|, |v|))$, mais ce n'est pas demandé ici.