Algorithmique Avancée - Devoir 2
Un correcteur orthographique
Enseignant responsable : Karim Nouioua

Jean-Baptiste Bonardo Éloi Perdereau

9 décembre 2012

Résumé

Dans le cadre du cours d'algorithmique avancé, il nous était demandé de programmer un correcteur orthographique selon une méthode bien spécifique décrite dans le cours. L'implémentation de cette méthode a requiert des structures de données. Bien que certaines existent dans différents langages, il a fallu les implémenter nous-mêmes et les utiliser pour l'algorithme de correction d'un mot.
Le langage utilisé ici est le C++. Les raisons sont assez simple :

1. connaissance assez avancée du langage
2. permet une grande généricité
3. sans que les performances n'en souffrent

Touts les identificateurs commencent par une majuscule. Ceci pour être sur de ne pas tomber sur une variable du C++.
La comparaison d'un pointeur se fait sans opérateurs, ainsi `if (Ptr)` vérifie simplement si `Ptr` pointe sur une donnée valide.

Table des matières

1 Distance de Levenshtein

On appelle distance de Levenshtein entre deux mots `MotOrig` et `MotDest` le "coût minimal" pour transformer le `MotOrig` en `MotDest` en effectuant les opérations élémentaires suivantes : substitution d'un caractère de `MotOrig` par un caractère de `MotDest`, ajout dans `MotOrig` d'un caractère de `MotDest`. suppression d'un caractère de `MotOrig`. À chacune de ces opérations est associé un coût. Pour simplifier, on attribuera à chaque opération un coût de 1 sauf dans le cas d'une substitution d'un caractère de `MotOrig` par le même caractère de `MotDest`, qui aura bien entendu un coût de 0. L'algorithme qui calcul la distance de Levenshtein utilise des matrices.

2 Coefficient de Jaccard

Les ensembles sont représentés par des `CLink`. On parcours le premier ensemble `A`, pour chaque élément, on parcours le deuxième ensemble `B` et ont test si les deux éléments sont identiques. Si oui, on incrémente la taille de l'intersection. En parcourant le premier ensemble, on a calculé sa taille, idem pour le deuxième ensemble (on l'a fait `taille de l'ensemble A` fois), donc à la fin du parcours de `A`, on doit diviser la taille de `B` calculé, par la taille de `A`. Cet algorithme est en $O(n^2)$, mais dans notre cas, on l'utilise sur des ensembles de trigrammes (5 éléments), donc c'est assez rapide.
On aurait pu appliquer un autre algorithme utilisant une table de hachage :
On met tout les éléments de `A` dans la table, puis pour chaque élément de `A`, on regarde s'il est dans la table de hachage précédemment remplie. Si oui, on incrémente la taille de l'intersection. Les tailles des deux ensembles sont calculés lors de leur parcours respectif. Cet algorithme est en $O(m+n)$ où `m` et `n` sont respectivement les tailles des ensembles A et B. On l'a pas implémenté car le premier algorithme est suffisamment efficace pour notre utilisation. Et aussi car pour stockée la table de hachage, on aurait du utiliser une `CHashMap`, et on ne sait pas comment hacher les éléments de la liste (générique).
On renvoie le résultat de l'opération suivante :

$$\frac{TailleDeL'Intersection}{|A| + |B| - TailleDeL'Intersection}$$

3 Structures de données implémentées

3.1 Liste simplement chaînée `CLink<T>`

La classe `CLink<T>` représente une liste simplement chainée générique. Elle possède les méthodes suffisantes et nécessaires à l'utilisation d'une telle liste.

1. Données membres : `m_Info` et `m_Suivant`
   Représentent respectivement l'information contenu dans le maillon, et un pointeur vers le maillon suivant.
2. Fonctions membres : getters et setters des données membres.

3.2 Table de hachage

3.2.1 Représentation d'une table de hachage

Une table de hachage est représenté par un tableau de pointeurs vers des listes chainées. Chaque maillon correspond à une entrée dans la table, c'est à dire une paire `<clé, valeur>`. Pour récupérer une entrée correspondant à une clé donnée, on calcule le hash ce cette clé. Ce hash nous donne l'indice dans le tableau où chercher l'entrée. C'est parce que plusieurs clés peuvent avoir le même hash qu'on

a une liste d'entrées à chaque indice. En fait, lorsqu'on veut récupérer une entrée, on parcours la liste à l'indice du hash en recherchant une pair qui a pour clé la clé cherchée. Et pour ajouter une entrée dans la table (ou modifier une valeur), on va de nouveau à l'indice du hash de la clé, et soit on ajoute la nouvelle entrée, soit on récupère l'entrée correspondant à la clé et on modifie sa valeur.

3.2.2 La classe `CHashMap<K,V>`

Cette classe sert pour représenter une table de hachage. Elle comporte deux paramètres de généricitée :

1. `K` : type de la clé
2. `V` : type de la valeur. Doit posséder un constructeur par défaut.

La classe a ensuite été défini ainsi :

1. Quelques `typedef`s qui vont bien :

```
typedef K                          Key_t;
typedef V                          Value_t;
typedef std::pair<Key_t, Value_t> Entry_t;
typedef CLink<Entry_t>             LinkPair_t;
typedef std::vector<LinkPair_t *> VLinkPair_t;
typedef nsUtil::IHash<Key_t>       Hashor_t;
```

2. Données statiques :

```
static const float    s_LoadFactor;
static const unsigned s_DfltInitialCapacity;
```

   (a) `s_LoadFactor` : taux de remplissage de la table à ne pas dépasser (il vaut 0.75). Par exemple, si on a une table de capacité 1000000 et 750000 éléments, alors à la prochaine insertion, il faudra doubler la capacité de la table et re-hacher toutes les entrées.

   (b) `s_DfltInitialCapacity` : capacité initiale par défaut, vaut 16.

3. Données membres

```
unsigned m_NbElem;
unsigned m_Threshold;   // if m_NbElem > m_Threshold, resize
VLinkPair_t m_V;
Hashor_t * m_Hashor;    // ptr car polymorphisme dans le constructeur
```

   (a) `m_NbElem` : nombre d'éléments dans la table

   (b) `m_Threshold` : seuil de remplissage : capacité * loadfactor

   (c) `m_V` : vecteur principal de la table. C'est là où sont stockés les éléments sous forme de liste chaînée de `std::pair<Key_t, Value_t>`.

   (d) `m_Hashor` : objet-fonction qui sert a hacher la clé. `IHash<T>` est une « interface » générique qui contient une fonction binaire qui prend deux arguments : la clé à hacher (de type `T`) et la capacité de la table, et renvoie le hash (entier non signé). Ce hash sera un indice valide car il est positif, et il est modulo de la capacité de la table ce qui correspond à la taille de `m_V`.
   C'est avec cette donnée membre qu'on peux avoir une table de hachage générique : le constructeur de `CHashMap` prend en paramètre un objet `IHash<Key_t>` (pointeur). Pour ce TP, une seule classe de hash a été défini : `CHashStr` qui dérive de `IHash<std::string` et implémente le hash d'une `std::string`.

4. Les fonctions membres :

```
void Resize (unsigned Cap)                                        throw ();
unsigned        GetNbElem   (void)                          const throw ();
unsigned        GetCapacity (void)                          const throw ();
void FillNbEntree (std::vector<unsigned> & VNbEntree) const throw ();
Value_t & operator [] (const Key_t & Key)                         throw ();
const Entry_t * Find  (const Key_t & Key)                   const throw ();
      Entry_t * Find  (const Key_t & Key)                         throw ();
iterator begin ()                                                 throw ();
iterator end   ()                                                 throw ();
```

   (a) `Resize` : redimensionne la table. C'est à dire, créé un vecteur `Vect` qui a deux fois la capacité de `m_V`. Puis hache toutes les clés de `m_V` et les met les éléments dans `Vect` (affectation de pointeur - très rapide).

   (b) `GetNbElem` et `GetCapacity` renvoient respectivement `m_NbElem` et `m_V.size()`.

   (c) `FillNbEntree` : inutile pour l'utilisation normale d'une table de hachage, on l'utilise pour vérifier la répartition des entrées dans la table. En fait, on rempli un vecteur (passé en paramètre résultat) où la valeur correspond au nombre de listes qui ont la taille de l'indice.
   e.g : `VNbEntree[`**`2`**`]` correspond au nombre de listes (dans `m_V`) qui ont **2** élément.

   (d) `operator []` : recherche dans la table l'élément qui a la clé passé en paramètre et renvoie sa valeur. Si l'élément n'est pas présent dans la table, alors il est créé et initialisé par défaut (d'où la nécessité d'avoir le constructeur par défaut pour `Value_t`). La valeur trouvée est renvoyée par référence, car ainsi on peux affecter cette valeur à une nouvelle valeur. Donc pour ajouter un élément dans la table de hachage, il faut utiliser cet opérateur.
   e.g

   ```
   Map["accueil"] = 3;
   ```

   affecte la valeur `3` à l'élément de clé `accueil`, ou, si cet élément n'existe pas dans la table, il est créé et sa valeur vaudra `3` après cette instruction. C'est dans cette fonction que on vérifie le taux de remplissage et qu'on redimensionne la table si nécessaire.

   (e) `Find` : cherche dans la table l'élément qui a la clé passé en paramètre et le renvoie (pointeur vers la paire). Si l'élément n'est pas trouvé, `Find()` renvoie un pointeur nul.

   (f) `begin` : renvoie un iterateur vers le premier élément non nul de `m_V`. Ou `end()` si la table est vide.

   (g) `end` : renvoie l'élément succédant le dernier élément de `m_V`. En fait, c'est juste un iterateur vers un pointeur nul.

5. Classe `iterator`.
   L'objectif ici est de permettre à l'utilisateur d'une `CHashMap` d'itérer cette map sur toutes ses entrées de la même manière qu'il le ferait sur une `std::map` :

```
CHashStr H;
CHashMap<string, int> Map (&H);
Map["un"]   = 1;
Map["deux"] = 2;

for (CHashMap<string, int>::iterator It (Map.begin()); It != Map.end(); ++It)
    cout << It->first << '\t' << It->second << endl;
```

   affiche

```
deux    2
un      1
```

Les entrées sont affichées dans l'ordre de leur stockage dans le vecteur `m_V` et non de leur affectation lors du remplissage de la map.
Pour arriver à ce résultat, on a créé une classe `iterator` interne à `CHashMap<K,V>` qui contient les fonctions de base d'un `input iterator` comme définit dans la STL `http://www.cplusplus.com/reference/iterator/`.

(a) Données membres :

i. `m_Map` : pointeur de la map qui est en train d'être itérée.

ii. `m_Pos` : position courante dans `m_Map->m_V`.

iii. `m_Iter` : pointeur vers une liste de paires, c'est l'élément. courant

(b) Fonctions membres :

i. `operator *` : renvoie la paire courante : `m_Iter->GetInfo()`.

ii. `operator ->` : appel l'`operator *`.

iii. `operator ==` et `operator !=` : comparent les `m_Iter`

iv. `operator ++` : fixe l'itérateur courant au prochain élément existant dans la table. Soit c'est le dernier élément de la liste, l'itérateur deviendra le prochain élément non nul dans le vecteur (en parcourant le vecteur à partir de `m_Pos+1`) soit c'est pas le dernier élément de la liste, dans ce cas l'itérateur devient l'élément suivant.

De plus, la classe `CHashMap` est ami à la classe `iterator`. Lorsque `begin()` est appelé sur une `CHashMap`, on renvoie un `iterator` sur le premier élément non-vide de `m_V`. Pour cela, on parcours `m_V` et on appel le constructeur de `iterator` avec les bons paramètres. Ce constructeur est privé, c'est pour cela qu'il faut ce lien d'amitié, car seul une `CHashMap` ne devrait pouvoir créer un `iterator`, et ce, uniquement avec `begin()`.

4 Le correcteur orthographique

Le but ici est de pouvoir "corriger" un mot donné en cherchant des mots (présent dans un dictionnaire) qui lui sont proches. On utilise une méthode qui manipule des trigrammes. Les trigrammes d'un mot donné est l'ensemble de ses sous-chaînes de trois caractères consécutifs.
e.g : les trigrammes de `accueil` : { `acc`, `ccu`, `cue`, `uei`, `eil` }

4.1 Remplissage des dictionnaires

Pour corriger un mot, nous avons besoin de deux tables de hachage :

1. Dictionnaire simple.
   clé = mot du dictionnaire
   valeur = n'importe quoi (on a mis `int`)
   La valeur n'a aucune importance car, lorsque le dictionnaire est rempli, pour savoir si un mot donné y est présent, on vérifie juste si l'entrée qui a comme clé le mot à cherché existe. Si oui, alors le mot est effectivement dans le dictionnaire, sinon, il n'y est pas.
   Ce dictionnaire est rempli simplement en parcourant un fichier dictionnaire et en créant les entrées correspondant à ses mots. On met une valeur arbitraire (qui ne prend pas trop de place) en valeur.

2. Dictionnaire dit "des trigrammes".
   clé : un trigramme
   valeur : liste chaînée de mots contenant le trigramme en clé
   e.g d'une entrée : `eui ; accueil -> cercueil -> fauteuil`
   Pour remplir cette table, on utilise l'algorithme suivant :

```
1 pour mot ← tout les mots du dico faire
2 |   pour trig ← tout les trigrammes de mot faire
3 |   |   liste ← valeur de l'entree correspondant a trig dans le dico des trigrammes;
4 |   |   ajouter mot a liste;
5 |   finpour
6 finpour
```

Les deux dictionnaire sont remplis dans la même fonction `RemplirDicosAvecFichier` qui parcours une seule fois le fichier dictionnaire et rempli les deux tables. Cette méthode ne permet pas de calculer le temps de remplissage des tables séparément.

4.2 Correction d'un mot

L'algorithme prend en paramètre d'entrée le mot à corriger et les deux dictionnaires (dico simple et dico des trigrammes). Elle prend aussi en paramètre résultat le vecteur des propositions à remplir.

Data : Mot, Dico, DicoTrig
Result : Props
1 **si** Mot *est dans* $Dico$ **alors**
2 | ajouter Mot a $Props$;
3 | **return** Props;
4 **finsi**
5 $Cpt \leftarrow map < string, unsigned >$;
6 **pour** $trig \leftarrow$ *chaque trigrammes de* Mot **faire**
7 | $liste \leftarrow DicoTrig[trig]$;
8 | **si** $liste = \emptyset$ **alors**
9 | | **continue**
10 | **sinon**
11 | | **pour** $MotTrigCommun \leftarrow$ *chaque mot de* $liste$ **faire**
12 | | | $++Cpt[MotTrigCommun]$;
13 | | **finpour**
14 | **finsi**
15 | **pour** $EntreeCpt \leftarrow$ *chaque entree de* Cpt **faire**
16 | | $TrigsEntree \leftarrow$ *liste des trigrammes de* $EntreeCpt.cle$;
17 | | **si** $EntreeCpt.valeur \geq NbTrigCommunMin$ **et** $Jaccard(TrigsEntree) > JaccardMin$ **alors**
18 | | | ajouter $EntreeCpt.cle$ a $Props$;
19 | | **finsi**
20 | **finpour**
21 **finpour**
22 trier $Props$ en comparant $Levenshtein(Props[i])$ a $Levenshtein(Mot)$;
23 garder uniquement les $NbPropsMax$ premières propositions;
24 **return** Props ;

Algorithme 1: Correction d'un mot

Les deux boucles **pour** des lignes 11 à 13 et 15 à 20 auraient pu être compressées en une seule. Mais de cette manière, l'algorithme est plus lisible, et la perte de performance est négligeable.

Voici quelques résultats :

```
Temps à la construction des deux dictionnaires : 960 ms
Temps pour la correction de quelques mots :
temps pour acceuil : 7 ms
temps pour baillonettes : 68 ms
temps pour boudiste : 10 ms

Bilan : 11650ms
269 mots lus :
171 bonnes corrections  : 63,6 %
236 dans la liste des suggestions  : 87,7
33 non corriges :12,3 %
```

4.3 Suggestions d'amélioration

On pourrait utiliser des digrammes, des trigrammes ou des quadrigrammes selon le mot à corriger.