Epelle : un logiciel de détection de fautes d'orthographe

Paul Zimmermann

Résumé. Ce rapport décrit l'algorithme utilisé par le programme `epelle` et son implantation dans le langage C. Ce programme permet de vérifier plus de 30000 mots par seconde sur une station de travail, avec un taux d'erreur nul, contrairement aux méthodes de hachage utilisées par `spell`. Le principe est d'utiliser des arbres digitaux, ce qui permet aussi un gain en espace par rapport à la liste de mots (de l'ordre de 5 pour le dictionnaire français). La création de l'arbre digital correspondant au dictionnaire français (près de 240000 mots) ne dure qu'une dizaine de secondes. Le même programme est directement utilisable pour d'autres langues, et même pour n'importe quelle liste de mots alphanumériques.

Epelle : a general spelling-checker program

Abstract. This report describes the algorithm used by the `epelle` program, together with its implementation in the C language. This program is able to check about 30000 words every second on modern computers, without any error, contrary to the Unix `spell` program which makes use of hashing methods and could thus accept wrong words. The main principle of `epelle` is to use digital trees (also called dictionary trees), which in addition reduces the space needed to store the list of words (by a factor of about 5 for the french dictionary). Creating a new digital tree for the french language (about 240000 words) takes only a dozen of seconds. The same program is directly usable for other languages, and more generally for any list of alphanumeric keys.

Epelle : un logiciel de détection de fautes d'orthographe

Paul Zimmermann
Inria Lorraine

Résumé

Ce rapport décrit l'algorithme utilisé par le programme `epelle` et son implantation dans le langage C. Ce programme permet de vérifier plus de 30000 mots par seconde sur une station de travail, avec un taux d'erreur nul, contrairement aux méthodes de hachage utilisées par `spell`. Le principe est d'utiliser des arbres digitaux, ce qui permet aussi un gain en espace par rapport à la liste de mots (de l'ordre de 5 pour le dictionnaire français). La création de l'arbre digital correspondant au dictionnaire français (près de 240000 mots) ne dure qu'une dizaine de secondes. Le même programme est directement utilisable pour d'autres langues, et même pour n'importe quelle liste de mots alphanumériques.

La détection de fautes d'orthographe est un cas particulier de la gestion de bases de données soumises à des requêtes du type I ("*ajouter cette donnée*") ou Q ("*cette donnée est-elle dans la base ?*"). La base de données est une liste de mots (dictionnaire) et en pratique les interrogations Q sont beaucoup plus nombreuses que les ajouts I, qui peuvent par exemple n'être faites que par la personne qui maintient à jour le dictionnaire.

La première section décrit l'utilisation du hachage dans les programmes de détection de fautes d'orthographe (par exemple la commande Unix `spell` pour l'anglais [3]). La seconde section introduit les arbres digitaux et montre comment construire efficacement l'arbre digital associé à un dictionnaire. La section 3 décrit le dictionnaire français utilisé par `epelle`, tandis que la section 4 fournit des résultats concernant différentes langues et montre que le programme `epelle` s'avère utile dans d'autres domaines que la vérification d'orthographe. Enfin, la section 5 reproduit les fichiers de l'implantation en C.

1 Utilisation du hachage

Les méthodes de hachage sont bien adaptées à des situations où l'on désire savoir rapidement si une donnée existe dans une table, et lorsque l'on tolère une certaine probabilité d'erreur. C'est précisément le cas de la détection de fautes d'orthographe : la donnée est un mot du texte à vérifier, la table est un dictionnaire, et l'on est prêt à accepter une faible marge d'erreur.

Le principe des méthodes de hachage est le suivant : on projette l'ensemble $\mathcal{D}$ des mots sur les entiers naturels par une fonction f appelée *fonction de hachage*, puis on se ramène à un ensemble fini de valeurs, par exemple en prenant la valeur de f modulo un certain entier N. L'information est donc contenue dans une table de N booléens :

for $i := 0$ **to** $N-1$
    $T[i] \leftarrow$ false
foreach $m \in \mathcal{D}$
    $T[f(m) \bmod N] \leftarrow$ true

La recherche est très simple, puisqu'il suffit de regarder le booléen associé au mot à tester :

$$T[f(m) \bmod N] = \text{false} \implies m \text{ n'est pas dans } \mathcal{D}$$
$$T[f(m) \bmod N] = \text{true} \implies m \text{ est peut-être dans } \mathcal{D}$$

Si l'on rejette les mots pour lesquels $T[f(m) \bmod N]$ vaut *false* et que l'on accepte ceux pour lesquels ce booléen vaut *true*, on obtient donc un détecteur de fautes tels que tous les mots signalés sont effectivement faux, mais qui "oublie" certains mots erronés.

Plus précisément, si α est le taux de remplissage de la table (c'est-à-dire le rapport entre le nombre de booléens qui valent *true* et N), et si la probabilité que $f(m) \bmod N$ égale i est la même pour tout $0 \leq i < N$, alors l'erreur commise par le programme ne dépasse pas $1 - \alpha$.

C'est ce procédé de hachage qui est utilisé par le programme UNIX `spell` : la commande `hashmake` convertit une liste de mots en la liste de leurs codes, des nombres de neuf chiffres :

```
% cat dico
auto
autobus
camion
train
voiture
% /usr/lib/spell/hashmake < dico | tee dico.codes
454524372
647545412
460442120
417177515
323215174
```

Puis la commande `spellin` permet de former une table à partir d'une liste de codes :

```
% /usr/lib/spell/spellin 5 < dico.codes > dico.hash
spellin: expected code widths = 26.146019
spellin: 5 items, 0 ignored, 0 extra, 5 words occupied
spellin: 32.000000 table bits/item, 3315.200000 table+index bits
```

Ensuite, la commande `spell` prend en argument une table de hachage compressée et une liste de mots, et renvoie les mots dont la fonction de hachage ne correspond à aucun mot du dictionnaire :

```
% cat texte
autobu
velo
auto
voiture
% spell -d dico.hash < texte
autobu
velo
voiture
```

On aperçoit au passage qu'il y a un problème avec la commande `spell` puisque le mot *voiture* qui est dans le dictionnaire n'a pas été accepté.

Les techniques de hachage ont été utilisées pour la détection de fautes d'orthographe principalement pour des raisons historiques. En effet, lorsqu'en 1982 McIlroy développe `spell`, il travaille sur un PDP 11 qui ne peut adresser plus de 64Ko de mémoire, et il veut stocker un dictionnaire de 30000 mots ! Pour comparaison, le dictionnaire anglais d'Unix (`/usr/dict/words`) comprend 25000 mots et utilise plus de 200Ko. Par conséquent, son principal souci est d'économiser la mémoire, tout en conservant un taux d'erreur faible. Cela est très facile avec le hachage puisque chaque mot n'utilise qu'un bit. Pour obtenir un taux d'erreur de 1 pour 4000, McIlroy calcule que la fonction de hachage doit avoir comme image $[1 \ldots 12 \cdot 10^7]$, ce qui nécessiterait 15Mo de mémoire !

L'astuce de McIlroy consiste à stocker non tous les bits de la table de hachage, mais uniquement les indices de ceux qui sont à 1. Chaque indice peut être codé sur 27 bits, donc 27×30000 bits suffisent, mais cela fait encore $100Ko$.

La seconde astuce consiste à stocker uniquement les différences entre deux indices consécutifs. Ainsi, en moyenne 13.6 bits par mot suffisent, soit un peu plus de 50Ko pour le dictionnaire de 30000 mots, soit moins de deux octets par mot !

Le programme `spell` vérifie 5000 mots en 30 secondes sur un Vax 750, soit 165 mots par seconde. Pour plus de détails sur `spell`, lire l'article captivant de Bentley [1].

La version actuelle de `spell` vérifie les 25144 mots de `/usr/dict/words` en 1.6s sur une station HP 9000/700, soit près de 16000 mots par seconde.

2 Utilisation d'arbres digitaux

Comme montré dans la section précédente, le hachage a été utilisé dans `spell` plus pour des raisons historiques de gain en espace mémoire que pour la recherche d'une réelle efficacité.

Une méthode beaucoup plus naturelle (et de plus exacte) consiste à imiter le mode de recherche dans un dictionnaire : on commence par chercher la première lettre du mot à vérifier, puis la seconde, et ainsi de suite. En général, lorsque les trois premières lettres ont été trouvées, le mot se trouve (ou devrait se trouver) dans la page cherchée.

A partir du dictionnaire, on construit donc un arbre n-aire donc chaque nœud contient une lettre et un booléen (figure 1). Chaque nœud représente le préfixe obtenu en partant de la racine,

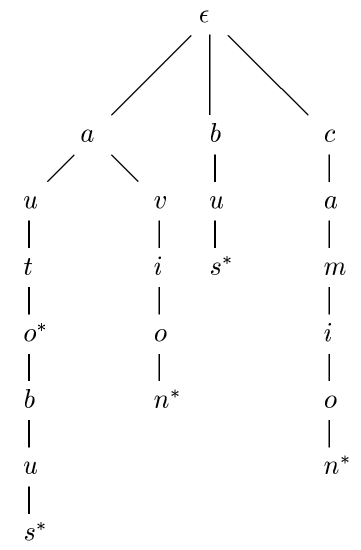

FIG. 1 – L'arbre digital n-aire du dictionnaire *auto, autobus, avion, bus, camion.*

par exemple le nœud contenant la lettre m représente le préfixe *cam*, et les booléens distinguent les préfixes qui sont des mots du dictionnaire.

Représenter un tel arbre n-aire est très coûteux en mémoire, puisqu'il faut prévoir autant de fils à chaque nœud que de lettres dans l'alphabet, soit 26 en français, sans compter les lettres accentuées ! Une première astuce classique consiste à transformer cet arbre n-aire en arbre binaire : les branches gauches correspondent à l'ajout d'une lettre au préfixe courant alors que les branches droites correspondent au remplacement de la dernière lettre du préfixe courant (figure 2). L'algorithme de recherche d'un mot $s_1 \dots s_l$ dans un tel arbre binaire est très simple :

Algorithme recherche($s_1 \dots s_l$).
{ *renvoie* true *si le mot est dans le dictionnaire,* false *sinon* }
    $i \leftarrow 1$ { *indice de la lettre à chercher* }
    $t \leftarrow$ racine de l'arbre digital
    tant que $i \leq l$
        tant que lettre(t) $\neq s_i$
            si t n'a pas de fils droit, renvoyer *false*, **sinon** $t \leftarrow$ fils droit de t
        si $i = l$, renvoyer booléen(t), **sinon** $i \leftarrow i + 1$

En effet cet algorithme n'est autre qu'un parcours dans un automate fini [2] : lorsque la lettre courante correspond à celle du nœud courant, on descend dans la branche gauche, sinon dans la branche droite.

Néanmoins l'espace mémoire utilisé reste important puisque chaque nœud doit contenir un caractère, un booléen et surtout deux pointeurs. L'arbre de la figure 2 contient 20 nœuds pour un dictionnaire de 25 lettres. La seconde amélioration consiste à compresser l'arbre digital binaire de façon à regrouper tous les sous-arbres identiques en un seul. Par exemple dans l'arbre de la

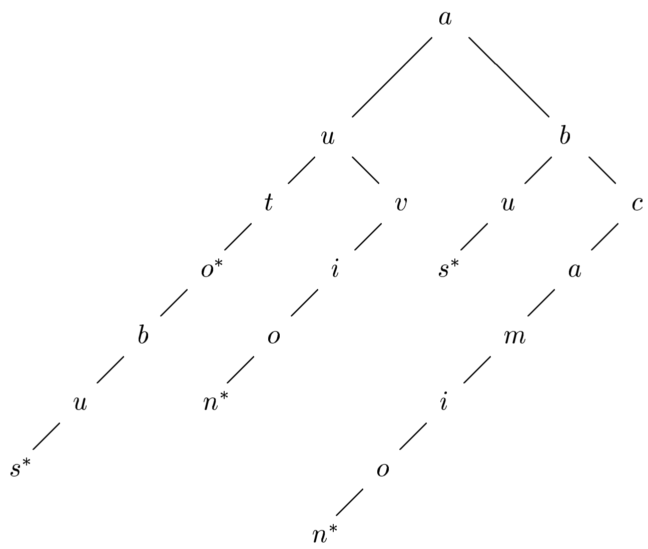

FIG. 2 – L'arbre digital binaire du dictionnaire *auto, autobus, avion, bus, camion.*

figure 2, le sous-arbre $-i-o-n^*$ apparaît en deux endroits, de même que le sous-arbre $-u-s^*$. On obtient donc l'arbre compressé de la figure 3. Reste à trouver un algorithme efficace pour

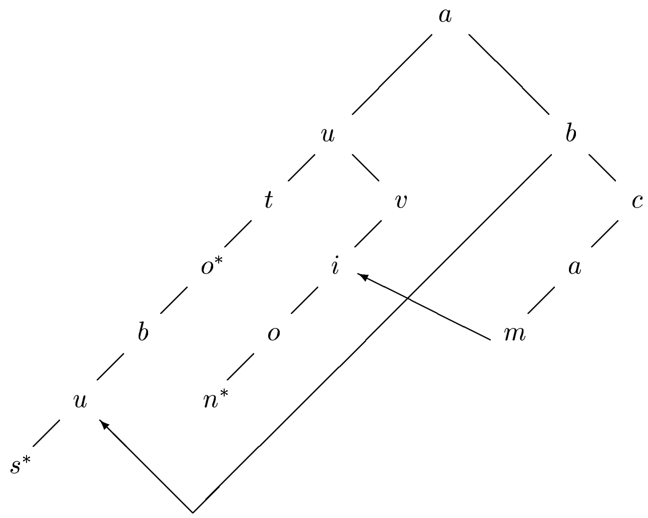

FIG. 3 – L'arbre digital compressé du dictionnaire *auto, autobus, avion, bus, camion.*

compresser un arbre digital binaire. L'algorithme naïf consiste à parcourir l'arbre de gauche à droite, en comparant chaque sous-arbre à tous ceux rencontrés jusque-là. Le nombre de nœuds étant n, le nombre de comparaisons entre sous-arbres est $O(n^2)$, chaque comparaison pouvant nécessiter $O(n)$ comparaisons, soit un coût de $O(n^3)$ dans le cas le pire. Cet algorithme naïf est donc trop coûteux pour les applications visées, où n peut valoir un million.

Afin de réduire le temps des comparaisons entre sous-arbres, on associe à chaque sous-arbre déjà traité un entier qui le caractérise. Ainsi, la comparaison de deux sous-arbres est réduite à une comparaison entre caractères, une autre entre booléens, et deux comparaisons entre entiers, ce qui réduit le temps de compression à $O(n^2)$ dans le cas le pire. D'où l'algorithme suivant :

Algorithme compactifie(t).
{ *renvoie l'entier associé à l'arbre* t }
  si t est vide, **alors** renvoyer 0

$j \leftarrow$ compactifie(t_{gauche})
$k \leftarrow$ compactifie(t_{droit})
$i \leftarrow$ cherche($j, k, t_{\text{car}}, t_{\text{bool}}$)
si $i < 0$ **alors** $i \leftarrow$ cree($j, k, t_{\text{car}}, t_{\text{bool}}$)
renvoyer i

où la fonction *cherche* recherche un quadruplet *(entier, entier, caractère, booléen)* dans la table des sous-arbres déjà traités, et la fonction *cree* insère un nouvel enregistrement dans cette table.

Pour réduire le coût de la fonction *cherche*, on utilise le hachage : à partir du quadruplet, on calcule un entier h qui va donner la place à laquelle doit se trouver ce quadruplet dans la table, s'il a déjà été créé. Au cas où cet emplacement est déjà occupé par un autre quadruplet, on essaie le suivant, et ainsi de suite jusqu'à trouver l'enregistrement cherché ou un emplacement vide, ce qui signifie qu'il n'a pas encore été crée (voir la section 5 pour plus de détails).

3 Le dictionnaire français

Le dictionnaire utilisé par `epelle` a été réalisé à partir de listes de mots réalisées par plusieurs personnes, notamment Pierre Lescanne et René Cougnenc. A partir d'une version intermédiaire du dictionnaire, Bruno Salvy a vérifié et ajouté un grand nombre de mots proposés par `ispell`. Enfin, de décembre 1992 à mai 1993, plus de 40.000 mots nouveaux ont été vérifiés par 66 personnes : François Thomasset, Philippe Robert, Anne Pelletier, Laurent Guillopé, Amedeo Napoli, François Paquet, France Guillou, André Girard, Michel Delval, Thérèse Hardin, Pascal Petit, Yannick Hebert, François Laissus, Lucile Cognard, François Velde, Salvatore Tabbone, Régis Curien, Martin Jourdan, Sylvain Petitjean, Joël Nicolas, Jeanine Souquières, Laurent Bloch, Olivier Plaut, Marc Bassini, Janick Taillandier, Jean-Marie Larchevêque, Damien Doligez, Corinne Loesel, Stéphane Bortzmeyer, Alex Stephenne, Steve Serocki, Régis Cridlig, Pierre Ferron, Philippe Mackowiak, David Perbost, Nicole Rocland, Christophe Muller, Michel Mauny, Bruno Marmol, Marie-Christine Haton, François Manchon, Malika Smaïl, Luc Maranget, Carole Le Bellec, Jean-Marie Kubek, Philippe Kerlirzin, Jean-Paul Haton, Jean-Luc Rémy, Jean-Alain Le Borgne, Michel Jacquot, Grégory Kucherov, Pierre Gagne, Paul Freedman, Fred Eichelbrenner, François Rouaix, Frédéric Chauveau, Arnaud Février, Frédéric Moinard, Jean Favre, Éric Turolla, Emmanuel Chailloux, Jay Doane, Didier Rémy, Alain Cousquer, Philippe Canalda, Guillaume Bres.

Le dictionnaire comprend près de 240000 mots, toutes formes comprises (pluriels, féminins des adjectifs, conjugaisons des verbes) et occupe près de 2.6Mo, soit une moyenne de 9.8 lettres par mot (un caractère marque la fin de chaque mot).

4 Quelques résultats

Nous montrons dans cette section que le programme `epelle` peut vérifier l'orthographe d'autres langues que le français, et même être utilisé dans d'autres applications, comme par exemple réaliser un test simple de primalité.

Des dictionnaires de plusieurs langues sont disponibles par *ftp* anonyme, par exemple sur les machines `ftp.ibp.fr` ou `ftp.eu.net`. Le programme `zpellin` de transformation d'un dictionnaire en un arbre digital compressé a été testé sur ces dictionnaires. La figure 4 montre que le temps de création de la table ne prend pas plus de 15 secondes sur une station HP 9000/700, ce qui est tout à fait acceptable, d'autant plus que la création d'une nouvelle table n'est pas fréquente. L'analyse détaillée du temps de fabrication de la table en fonction du nombre de mots du dictionnaire montre une variation linéaire, avec une constante variant entre 10000 et 30000 mots par seconde. La figure 4 montre aussi que les dictionnaires français et italiens donnent un meilleur taux de compression (environ 5) que par exemple l'allemand ou l'anglais (environ 2). Dans tous les cas cependant, la taille de la table est au moins deux fois inférieure à celle du dictionnaire.

Le programme `epelle` procède comme suit : dans un premier temps, la commande `detex` extrait les mots du fichier donné (et élimine les commandes (La)TeX avec `epelle -latex`), puis

langue	mots	taille dico	nœuds avant compression	nœuds après compression	taille table	temps de création (HP 9000/700)
allemand	160087	2.1Mo	499699	150839	0.9Mo	15s
anglais	51899	0.5Mo	130155	43174	0.3Mo	2.6s
français	239211	2.6Mo	523703	84286	0.5Mo	12s
hollandais	148602	1.7Mo	427882	106648	0.6Mo	8.8s
italien	60541	0.6Mo	111822	19159	0.1Mo	2.2s
norvégien	61844	0.6Mo	156548	57879	0.3Mo	2.9s

FIG. 4 – Compression par `epelle` de plusieurs dictionnaires

ces mots sont triés avec `sort -u`, enfin ils sont vérifiés avec `zpellout`. La figure 5 montre que le temps de vérification proprement dit (colonne `zpellout`) ne représente plus qu'une faible partie du temps total de traitement du fichier (colonne `epelle`).

fichier	nombre total de mots	mots différents	`epelle`	`zpellout`
dictionnaire français	239211	239211	71s	11s
`/usr/dict/words`	25144	25144	7s	1s
ce document	2488	754	1s	0.3s

FIG. 5 – Temps de vérification de plusieurs fichiers

Plus généralement, les commandes `zpellin/zpellout` peuvent vérifier l'appartenance à n'importe quel ensemble de chaînes alphanumériques. Par exemple, fabriquons à l'aide de Maple un fichier contenant les 100000 premiers nombres premiers :

```
> writeto(`primes`);
> for i to 100000 do lprint(ithprime(i)) od:
> quit
```

puis transformons le fichier `primes`, qui utilise 0.7Mo, en un arbre digital compressé :

```
% zpellin < primes > primes.z
% time zpellout primes.z < primes
1.97u 0.14s 0:02.11 100.0%
```

Le fichier `primes.z` ne fait que 0.3Mo, et permet de tester la primalité d'une liste d'entiers inférieurs à 1299709 en 20 microsecondes par nombre ! Notons d'ailleurs en ce qui concerne la taille mémoire que `zpellin` est compétitif vis-à-vis de `gzip` :

```
-rw-rw-r--   1 zimmerma eureca    341868 Jul 22 17:39 primes.z
-rw-rw-r--   1 zimmerma eureca    243061 Jul 22 17:39 primes.gz
-rw-rw-r--   1 zimmerma eureca    610196 Jun 28 18:02 words.french.gz
-rw-rw-r--   1 zimmerma eureca    505716 Jul 22 15:19 words.french.z
```

Une autre application possible serait la vérification des mots de passe par rapport à une liste de mots interdits, mais nous abordons là un sujet sensible qu'il vaut mieux éviter !

5 Implantation en C

Les fichiers ci-dessous ainsi que le programme `epelle` sont accessibles par *ftp* anonyme sur la machine `ftp.inria.fr`.

5.1 Le fichier epelle.h

Ce fichier est utilisé par zpellin et zpellout (cf sections suivantes).

```
#include <stdio.h>
#include <ctype.h>

#define MAX_NODES 160000             /* maximal number of nodes after compaction */
#define PRIME 160001                /* must be greater than MAX_NODES */

#define I(a) ((unsigned int) a)
```

5.2 La fonction zpellin

Cette fonction transforme une liste de mots en une table représentant l'arbre digital binaire compressé correspondant (cf section 2).

```
/* Usage: zpellin < liste_mots > table */

#include "epelle.h"

#define C(a) ((char) a)
#define LOW(a) (a & 0xFF)
#define HIGH(a) ((a & 0xFF00)>>8)
#define VHIGH(a) ((a & 0x70000)>>16)
#define ERROR(s) {fprintf(stderr,"s\n"); exit(1);}

typedef struct NOEUD {
    char c,existe;
    struct NOEUD *fils,*frere;
} noeud;

noeud type_noeud[MAX_NODES];
int hashtab[PRIME];
int nb_noeuds=0,nb_types=0,traites=0;

noeud* nouveau_noeud(cc)
char cc;
{
    noeud *t;

    t = (noeud*) malloc(sizeof(noeud));
    if (t==0) ERROR(ran out of memory)
    t->c = cc;  t->existe = 0;  t->fils = t->frere = NULL; ++nb_noeuds;
    return(t);
}

main()
{
    noeud *root,*t;
    char cc;
    int nb_mots=0,i;

    t = root = nouveau_noeud('?');
    while ((cc = getchar()) != EOF)
        if ((unsigned)cc>' ')
          if (t->fils != NULL) {
           t = t->fils;
           while (t->c != cc)
```

```
                if (t->frere != NULL) t = t->frere;
                else t = t->frere = nouveau_noeud(cc);
           }
           else t = t->fils = nouveau_noeud(cc);
         else { t->existe = 1;  ++nb_mots;  t = root; }
    fprintf(stderr,"%d words, %d nodes, ",nb_mots,nb_noeuds);
    for (i=0;i<PRIME;i++) hashtab[i]=0;
    compactifie(root);
    fprintf(stderr,"%d nodes after compaction\n",nb_types);
    out();
    exit(0);
}

int hash (fi,fr,c,b)
int fi,fr;
char c,b;
{
    return(I(16061*fi + 16063*fr + 16067*I(c) + 16069*I(b)) % PRIME);
}

int cherche(fi,fr,c,b)
int fi,fr;
char c,b;
{
    int h,i;

    h = hash(fi,fr,c,b);
    while (hashtab[h] != 0) {
        i = hashtab[h];
        if (I(type_noeud[i].fils) == fi)
            if (I(type_noeud[i].frere) == fr)
                if (type_noeud[i].c == c)
                    if (type_noeud[i].existe == b)
                        return(i); /* trouve' */
        h = (h+1) % PRIME;
    }
    hashtab[h] = ++nb_types;
    return(-1); /* non trouve' */
}

int compactifie(t)
noeud *t;
{
    int type_frere, type_fils,i;

    if (t == NULL) return(0);
    else {
        traites++;
        type_frere = compactifie(t->frere);  type_fils = compactifie(t->fils);
        i = cherche(type_fils,type_frere,t->c,t->existe);
        if (i != -1) return(i);   /* sous-arbre de'ja` rencontre' */
        i = nb_types;
        if (i>=MAX_NODES)
            {fprintf(stderr,"too many nodes : %d/%d\n",traites,nb_noeuds); exit(1);}
        type_noeud[i].c = t->c;  type_noeud[i].existe = t->existe;
        type_noeud[i].fils = (noeud*) type_fils;
        type_noeud[i].frere = (noeud*) type_frere;
        return(i);
```

```
    }
}

out() /* e'crit la table compacte'e en stdout   */
/* format : 6 octets par enregistrement        */
/*      0 : caracte`re                         */
/*      1 : boole'en + 3 bits + 3 bits vhigh    */
/*      2 : fils low                           */
/*      3 : fils high                          */
/*      4 : frere low                          */
/*      5 : frere high                         */
/* la racine est le dernier enregistrement      */
{
    int i,j,k;

    for (i=1; i<=nb_types; i++) {
        putchar(type_noeud[i].c);
        k = I(type_noeud[i].existe) + (VHIGH(I(type_noeud[i].fils))<<1)
            + (VHIGH(I(type_noeud[i].frere))<<4);
        putchar(C(k));
        putchar(C(LOW(I(type_noeud[i].fils))));
        putchar(C(HIGH(I(type_noeud[i].fils))));
        putchar(C(LOW(I(type_noeud[i].frere))));
        putchar(C(HIGH(I(type_noeud[i].frere))));
    }
}
```

5.3 La fonction zpellout

Cette fonction vérifie une liste de mots par rapport à une table créée par zpellin.

```
/* Usage: zpellout table < liste_mots > mots_errone's */

#include "epelle.h"

#define MAXLENGTH 256                   /* longueur maxi des mots */
#define CTOI(a) ((a<0) ? a+256 : a)

typedef struct NOEUD {
    char c,existe;
    unsigned int fils,frere;
} noeud;

noeud T[MAX_NODES];
int racine;

main(argc,argv)
int argc;
char* argv[];
{
    FILE *f,*fopen();
    char c,mot[MAXLENGTH],c1,c2;
    int i=0;

    /* entre'e de la table */
    f = fopen(argv[1],"r");
    while ((c = getc(f)) != EOF) {
        T[++i].c = c;
        c =  getc(f);
```

```
        T[i].existe = I(c) & 0x1;
        c1 = getc(f); c2 = getc(f);
        T[i].fils = CTOI(c1) + (CTOI(c2) << 8);
        c1 = getc(f); c2 = getc(f);
        T[i].frere = CTOI(c1) + (CTOI(c2) << 8);
        T[i].fils += (I(c) & 0xE) << 15;
        T[i].frere += (I(c) & 0x70) << 12;
    }
    racine = i;
    /* boucle de de'tection de mots errone's */
    i = 0;
    while ((c = getchar()) != EOF)
        if ((unsigned)c>' ') mot[i++] = c;
        else {
            mot[i] = 0;
            /* un mot est accepte' lorsqu'en mettant toutes ses lettres
               en minuscules, on trouve un mot du dictionnaire */
            if (!valide(mot)) printf("%s\n",mot);
            i = 0;
            }
    exit(0);
}

int valide(s)
char *s;
{
    int i,t;
    char c;

    i=0; t=racine;
    while ((c = tolower(s[i])) != 0) {
        if (T[t].fils == 0) return(0); /* non valide */
        t = T[t].fils;
        while (T[t].c != c)
            if (T[t].frere == 0) return(0);
    else t = T[t].frere;
        i++;
    }
    return(I(T[t].existe)); /* 0 ou 1 */
}
```

Remerciements. C'est avec Luc Albert qu'une première version du logiciel `epelle` a été réalisée en 1988. Merci aussi à Bruno Salvy qui a complété systématiquement le dictionnaire, et à Philippe Robert qui a relu avec perspicacité une version préliminaire de ce rapport.

Références

[1] Bentley, J. A. A spelling checker. *Commun. ACM 28*, 5 (May 1985), 456–462.

[2] Lucchesi, C. L., and Kowaltowski, T. Applications of finite automata representing large vocabularies. Relatório Técnico 9/91, Instituto de Matemática, Universidade Estadual de Campinas, São Paulo, Brasil, 1991.

[3] McIlroy, M. D. Development of a spelling list. *IEEE Transactions on Communications 30*, 1 (Jan. 1982), 91–99.