

Algorithmes et structures de données,
laboratoire n° 4
AdaSpell
un correcteur d'orthographe.

Daniel Lifschitz et Nicolas Seriot,
classe EI3B, EIVD, Yverdon.

29 février 2004

Résumé

Ce laboratoire concerne le développement d'un correcteur d'orthographe. Le programme, écrit en Ada95, permet de charger un dictionnaire, de repérer les mots d'un texte qui ne font pas partie du dictionnaire, pour soit y apporter des corrections de manière interactive, soit rediriger les propositions vers un fichier.

L'implémentation met l'accent sur une bonne utilisation de la mémoire et la rapidité de l'exécution, en définissant un arbre particulier pour stocker de gros dictionnaires et retrouver l'information rapidement.

Table des matières

1	Spécification	3
1.1	Introduction	3
1.2	Objectif	3
2	Conception	4
2.1	Structure de données	4
2.1.1	L'arbre AVL	4
2.1.2	L'arbre n-aire	5
2.1.3	L'arbre n-aire d'articles et de pointeurs sur tableaux de pointeurs	7
2.1.4	L'arbre n-aire d'articles à discriminant sans valeur par défaut	8
2.1.5	Considérations sur les pragmas	9
2.1.6	Le dictionnaire	11
2.1.7	Les propositions	12
2.2	Architecture du programme	12
2.2.1	AdaSpell	13
2.2.2	Correcteur_Orthographique	13
2.2.3	Interface_Utilisateur	14
2.2.4	Chrono	15
3	Guide utilisateur	17
3.1	Introduction	17
3.2	Les fichiers	17
3.3	Utilisation du programme	17
3.3.1	La correction en mode interactif	18
3.3.2	La correction en mode automatique	18
4	Tests	19
5	Journal de travail	20
6	Conclusion	24

Liste des algorithmes

1	Insérer un mot dans le dictionnaire	14
2	Savoir si un mot se trouve dans le dictionnaire	15
3	Contrôler un mot	16

1 Spécification

Remarque

Pour établir cette spécification, nous nous sommes basés sur ce qui avait été demandé aux étudiants de Jörg Kienzle à l'EPFL, à l'occasion d'un projet similaire (voir [1]).

1.1 Introduction

La recherche d'une information parmi un grand nombre d'éléments est quelque chose d'omniprésent en informatique. Au fur et à mesure que le nombre d'éléments croît, le choix de bonnes structures de données et des algorithmes adaptés est essentiel pour les performances. Parallèlement, l'occupation de la mémoire devient rapidement problématique.

1.2 Objectif

Nous allons tenter d'écrire un correcteur d'orthographe capable de traiter des textes longs, de l'ordre de plusieurs dizaines de pages, en un temps le plus court possible. Nous allons essayer de faire en sorte que l'on puisse utiliser un dictionnaire composé de plusieurs centaines de milliers de mots, tout en économisant la mémoire au maximum.

Il s'agira de donner deux fichiers au programme, une liste de mots qui constituera le dictionnaire d'une part, un texte à corriger d'autre part. Le résultat de l'analyse sera enregistré dans un troisième fichier.

Il devra également être possible de corriger un fichier texte de manière interactive.

Les mots du dictionnaire sont composés des caractères minuscules 'a' à 'z'. En fait, le fichier du dictionnaire peut contenir des majuscules, mais elles seront converties en minuscules lors de l'insertion du mot dans le dictionnaire. Le texte à analyser, lui, peut contenir n'importe quel caractère. Cependant, les caractères qui ne sont pas des lettres seront considérés comme des caractères de séparation.

Le programme devra déterminer si un mot fait partie du dictionnaire ou non. Il devra aussi, le cas échéant, proposer une liste de mots alternatifs. Plus précisément, le programme devra proposer des corrections aux fautes suivantes (nous prenons ici l'exemple du mot *eivd*) :

- l'utilisateur a tapé une lettre de trop – *eizvd* ;
- l'utilisateur a oublié une lettre – *evd* ;
- l'utilisateur s'est trompé en tapant une lettre – *eizd* ;
- l'utilisateur a échangé deux lettres – *evid*.

2 Conception

2.1 Structure de données

Nous avons imaginé plusieurs structures de données pour stocker le dictionnaire, la plupart à base d'arbre. Nous avons construit plusieurs prototypes dont nous avons pu évaluer les performances. Au final, nous avons retenu un arbre n-aire un peu particulier.

2.1.1 L'arbre avl

La première structure que nous avons imaginée est un arbre AVL dont la clé est une chaîne non contrainte (`Unbounded_String`). En effet, l'arbre AVL est à première vue la structure idéale pour la recherche rapide, puisque la complexité de la recherche de ses éléments est en $O(\log(n))$. Nous avons donc construit un prototype à partir du paquetage `Arbre_Avl_G` (voir [5]).

Nous nous sommes rapidement rendu compte de trois problèmes :

D'une part, nous stockions plusieurs fois la même information. Exemple : après l'insertion des mots `fire`, `fish` et `fine`, l'arbre est tel que représenté à la figure 1.

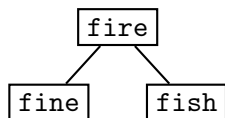


FIG. 1 – arbre AVL de chaînes non contraintes

Sur cette figure, on voit que la sous-chaîne de caractère "fi" est stockée trois fois. On imagine alors la quantité de mémoire gaspillée dans le cadre de l'utilisation d'un grand dictionnaire.

D'autre part, nous avons simplement constaté que l'insertion est beaucoup plus lente dans cet arbre que dans notre deuxième prototype, dont la description suit. En effet, l'arbre subit de nombreux rééquilibrages pour conserver ses propriétés, d'autant plus que les listes de mots que l'on insère pour remplir le dictionnaire sont généralement triées.

Enfin, dans cette structure, les mots "compute" et "computer" sont stockés dans deux nœuds séparés. Quel gaspillage !

2.1.2 L'arbre n-aire

L'idée est de construire un arbre dont les nœuds représentent des lettres de l'alphabet. Le parcours de l'arbre depuis la racine revient à déterminer si une suite de lettre constitue un mot. Une schéma simplifié de cette structure est donné à la figure 2. Cette figure met bien en évidence le gain de place par rapport à l'arbre AVL présenté à la figure 1.

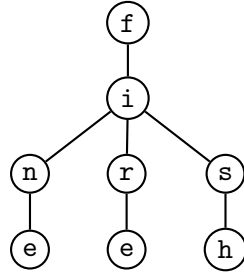


FIG. 2 – arbre n-aire

Bien sûr, il faut pouvoir « s'arrêter en chemin » si une suite de caractères forme un mot, même s'il existe d'autres mots qui commencent par les mêmes caractères. Par exemple, pour savoir si le mot "compute" existe, on va parcourir l'arbre mais, arrivé au 'e', un pointeur pointe sur le 'r', car "computer" est aussi mot. Comment savoir si "compute" est un mot ? et si "comput" en est un ?

Nous avons résolu cette question en introduisant dans le nœud une valeur booléenne qui indique si le chemin parcouru depuis la racine jusqu'au nœud courant forme un mot ou non.

Voici le code du type de données :

```

-- les caractères qui vont composer les mots du dictionnaire
subtype T_Caractere is Character range 'a'..'z';

type T_Tab_Cellules;

type T_Ptr_Tab_Cel is access T_Tab_Cellules;

type T_Cellule is
  record
    Existe      : Boolean := False;
    Terminal    : Boolean := False;
    Next        : T_Ptr_Tab_Cel;
  end record;

type T_Tab_Cellules is array(T_Caractere) of T_Cellule;

```

On observe que l'introduction du dictionnaire *web2*[4]¹ dans cette structure prend 6 secondes sur un processeur PPC G4/667 et déclare 566 913 nœuds. Il est intéressant de constater que, contrairement à l'arbre AVL pour lequel chaque nouveau mot entraîne la création d'un nouveau nœud, ici plus le nombre de mot croît et moins il est probable qu'il faille créer un nouveau nœud pour créer ce mot². Notre structure de donnée est donc bien adaptée à de grands dictionnaires.

Dans les faits, le grand avantage de cette arbre est la complexité des algorithmes d'insertion et de recherche : il est seulement en $O(\log(n))$ avec n désignant la longueur du mot concerné et cette complexité est toujours garantie. Pour s'en convaincre, il suffit de rechercher un des trois mots présentés à la figure 2.

Lors des tests, nous avons observé qu'il était bien plus rapide de charger un dictionnaire dans cette structure que dans un arbre AVL. Malheureusement, le gain en mémoire, si il est bien réel, n'est pas suffisant. Pour donner un ordre d'idée, le stockage du dictionnaire *web2* demande 114 Mo de mémoire vive. C'est trop ! Comment pouvons-nous occuper moins de place ?

En fait, quand nous déclarons une cellule, nous voulons stocker deux informations booléennes et un pointeur. Les deux informations booléennes pourraient tout-à-fait tenir sur 2 bits, mais le compilateur, par défaut et pour des raisons de temps d'accès à la mémoire, les stocke sur deux mots mémoire, constitués chacun de plusieurs bits. Quand nous déclarons 566 913 nœuds, que chaque nœud contient 26 cellules et que chaque cellule contient deux valeurs booléennes, c'est près de 30 millions de mots mémoire qui sont occupés pour stocker 30 millions de bits. On sent combien il serait intéressant de pouvoir compresser l'espace occupé par ces millions de cellules, quitte à perdre un peu en temps d'accès.

Le pragma `Pack`³ permet de compresser un tableau, un enregistrement ou un agrégat. En compressant `T_Cellule`, la mémoire occupée par le programme, après qu'il a chargé le dictionnaire *web2*, passe de 114 Mo à 80 Mo. D'après nos tests, le temps perdu lors de l'accès est si faible qu'il est difficile à mesurer.

Cependant, cette structure n'est pas vraiment satisfaisante, car tous les nœuds de l'arbre contiennent un tableau de 26 pointeurs. Or, ces pointeurs ne sont utiles que pour les nœuds qui ne sont pas des feuilles. Et comme notre arbre est peu profond (moins de

¹Notons que ce dictionnaire, contrairement à celui qui l'on utiliserait dans le cadre d'une utilisation courante du programme, ne contient pas les pluriels ni les formes conjuguées. Cela se traduit par une utilisation de mémoire supérieure, à nombre de mots égal, par rapport à un dictionnaire que l'on choisirait pour effectuer des corrections.

²En effet, si le mot à insérer est contenu dans un autre, on ne crée pas de nouveau nœud. Par exemple, si le dictionnaire comporte le mot *funky*, l'ajout des mots *funk* et *fun* ne modifiera pas la taille de l'arbre.

³Voir [2], chapitre 7 : « Optimizing Your Project ».

30 niveaux) mais très large, puisque l'alphabet utilisé comprend 26 lettres, il y a un très grand nombre de feuilles et donc un gaspillage important de la mémoire.

2.1.3 L'arbre n-aire d'articles et de pointeurs sur tableaux de pointeurs

La structure précédente déclare 26 pointeurs NULL sur chacune des feuilles et, comme on l'a vu, les feuilles sont très nombreuses. On contourne cet inconvénient en introduisant une distinction dans la composition des nœuds selon qu'ils sont des feuilles ou non. Si le nœud est une feuille, alors il n'a pas de tableau de 26 pointeurs.

Nous avons réalisé une première implémentation de cette structure, en plaçant sur chaque nœud un pointeur sur un tableau de 26 pointeurs. Ainsi, les feuilles n'ont plus qu'un seul pointeur NULL plutôt que 26. Le type de données est déclaré comme suit :

```
-- les caractères qui vont composer les mots du dictionnaire
subtype T_Caractere is Character range 'a'..'z';

-- un élément d'un noeud
type T_Element is
  record
    Existe      : Boolean := False; -- Indique si la lettre existe
    Terminal    : Boolean := False; -- Désigne un mot complet
  end record;

pragma Pack(T_Element);

type T_Tab_Elements is array(T_Caractere) of T_Element;

pragma Pack(T_Tab_Elements);

type T_Noeud;

type T_Ptr_Noeud is access T_Noeud;

type T_Tab_Ptr_Noeuds is array(T_Caractere) of T_Ptr_Noeud;

type T_Ptr_Next is access T_Tab_Ptr_Noeuds;

type T_Noeud is
  record
    Elements    : T_Tab_Elements;
    Next        : T_Ptr_Next := null;
  end record;

pragma Pack(T_Noeud);
```

Le gain de mémoire est réel, puisque l'on passe de 80 Mo à 54 Mo⁴, toujours avec *web2*. Sur un total de 566 913 nœuds, 411 712 ne sont pas des feuilles. On économise donc 155 201 tableaux de 26 pointeurs.

Cependant, chaque nœud déclare un pointeur supplémentaire, celui qui pointe sur son tableau de pointeur. D'autre part, le code devient peu lisible. Comment faire mieux ?

2.1.4 L'arbre n-aire d'articles à discriminant sans valeur par défaut

Ada met à disposition la notion d'article à discriminant ; c'est la solution que nous avons finalement retenue. Un discriminant indique si le nœud est un feuille. Si c'est le cas, alors le nœud, qui est un enregistrement, implémente uniquement un tableau contenant la description des cellules. Dans le cas contraire, le nœud comporte en plus un tableau de 26 pointeurs désignant ses fils.

Le discriminant n'a pas de valeur par défaut, ce qui signifie que la valeur du discriminant ne peut changer pendant l'exécution. Cela répond à notre souci d'économie de la mémoire, puisque seule la mémoire nécessaire est réservée. Par contre, l'algorithme d'insertion sera un peu plus compliqué.

L'inconvénient, car il y en a un, c'est la performance. D'une part, la structure de l'article introduit une indirection supplémentaire qui pénalise la recherche notamment. D'autre part, l'algorithme d'insertion est plus lourd ; l'ajout d'un nœud demande de passer par la déclaration d'un élément temporaire, de copier le tableau des éléments, de descendre la feuille... (voir l'algorithme 1 page 14).

Définition de la structure :

```
-- les caractères qui vont composer les mots du dictionnaire
subtype T_Caractere is Character range 'a'..'z';

-- un élément d'un noeud
type T_Element is
  record
    Existe      : Boolean := False; -- Indique si la lettre existe
    Terminal    : Boolean := False; -- Désigne un mot complet
  end record;

pragma Pack(T_Element);

-- tableau comportant un élément par lettre de l'alphabet.
type T_Tab_Elements is array(T_Caractere) of T_Element;

-- prédéclaration
type T_Noeud;
```

⁴Compilation réalisée en utilisant le pragma Pack.

```

type T_Ptr_Noed is access T_Noed;

-- tableau comportant 26 pointeurs sur les fils
type T_Tab_Ptr_Noeds is array(T_Caractere) of T_Ptr_Noed;

-- un noed est soit une feuille (il n'a pas de fils) soit intermédiaire.
type T_Noed (Feuille : Boolean) is
  record
    Elements : T_Tab_Elements; -- les 26 éléments
    case Feuille is
      when True => null;
      when False => Next : T_Tab_Ptr_Noeds; -- les 26 fils
    end case;
  end record;

type T_Dico(Dico_Perso : Boolean := False) is
  record
    Nb_Mots      : Natural := 0; -- nb de mots contenu dans le dico
    Nb_Cel       : Positive := 1; -- nb de noeuds présent dans l'arbre
    Nb_Car       : Natural := 0; -- nb de caractères dans le dico.
    Racine       : T_Ptr_Noed := new T_Noed(False); -- racine de l'arbre
    case Dico_Perso is
      when True => Fichier : Ada.Strings.Unbounded.Unbounded_String;
      when False => null;
    end case;
  end record;

```

2.1.5 Considérations sur les pragmas

Avec cette dernière structure, et sans l'utilisation de clauses pragma Pack, un nœud comportant des fils occupe 1 280 bits en mémoire et une feuille 448, soit 16 de moins que la structure déclarant un pointeur sur un tableau de pointeurs⁵ (voir 2.1.3 page 7).

Il est possible de réduire la taille d'un nœud et d'une feuille à respectivement 888 et 56 bits, soit 392 bits de moins en incluant les trois directives de compilation suivantes :

```

pragma Pack(T_Element);
pragma Pack(T_Tab_Elements);
pragma Pack(T_Noed);

```

Si on multiplie ces 392 bits par les 566 913 nœuds nécessaire au stockage du dictionnaire *web2*, on remarque qu'il est possible d'économiser un peu plus de 26 Mo de mémoire.

⁵Un pointeur utilise 32 bits en mémoire alors que le type boolean utilisé pour le discriminant n'en utilise que 16.

```

$ gnatmake correcteur_orthographique_t.adb
gcc -c correcteur_orthographique_t.adb
gcc -c correcteur_orthographique.adb
+=====GNAT BUG DETECTED=====+
| 3.3 20040222 (GNAT build 1650) (powerpc-unknown-darwin) GCC error:      |
| in store_field, at expr.c:5556                                         |
| Error detected at correcteur_orthographique.adb:126:44                 |
| Please submit a bug report; see http://gcc.gnu.org/bugs.html. |
| Include the entire contents of this bug box in the report.             |
| Include the exact gcc or gnatmake command that you entered.           |
| Also include sources listed below in gnatchop format                   |
| concatenated together with no headers between files.                   |
+=====+
Please include these source files with error report

correcteur_orthographique.adb
correcteur_orthographique.ads
chrono.ads

compilation abandoned
gnatmake: "correcteur_orthographique.adb" compilation error

```

FIG. 3 – Bug du compilateur GNAT.

Malheureusement, aussi bien le compilateur GNAT que celui d'ObjectAda comportent un bug qui se manifeste occasionnellement lorsque l'on copie le tableau des éléments d'une feuille dans un nœud qui n'en est pas une. Dans le cas du compilateur GNAT version 3.15p, le bug est vraiment pernicieux, car la copie ne s'effectue pas correctement, mais le programme continue son exécution. Dans le cas d'objectAda version 7.2.2, l'exception `Constraint_Error` est levée lors du chargement d'un dictionnaire après l'avoir vidé de la mémoire. Pour finir, sur Mac OS X, le GNAT version 5.00w ne parvient pas à compiler le fichier. Le message d'erreur est donné à la figure 3.

Nous avons donc effectué de nombreux essais avec le compilateur ObjectAda. La figure 4 indique les résultats⁶ après chargement en mémoire du dictionnaire *web2* qui, rappelons-le, génère 566 913 nœuds dans le dictionnaire.

On constate d'une part que la directive de compilation `Pragma Pack(T_Noead)` n'apporte aucune compression supplémentaire des données. D'autre part, la variante la plus économe en mémoire est aussi la plus rapide. On peut donc en conclure que non seulement l'instruction `Pragma Pack(T_Noead)` n'apporte aucune amélioration, mais en plus elle lève une exception de manière injustifiée.

⁶Mesures réalisées sur un Pentium IV 2.2 GHz.

Pragma Pack			Durée chargement	Mémoire utilisée
T_Element	T_Tab_Elements	T_Noeud	<i>en ms</i>	<i>en Ko</i>
			671	78 036
✓			661	64 729
	✓		671	78 036
		✓	671	78 036
✓	✓		581	51 404
✓		✓	661	64 720
	✓	✓	691	78 036
✓	✓	✓	Constraint_Error	

FIG. 4 – Incidence des différentes combinaisons de pragmas.

Malheureusement, ce qui est vrai pour le compilateur ObjectAda ne l'est pas pour le GNAT ; ce dernier n'arrive pas copier correctement le tableau des éléments d'une feuille vers une autre si on utilise la directive de compilation `Pragam Pack(T_Tab_Elements)`. C'est pourquoi cette ligne est commentée dans le code source. Il est bien évidemment possible – et recommandé – de la décommenter si le programme est compilé avec ObjectAda.

2.1.6 Le dictionnaire

Un dictionnaire est un enregistrement dont un des champs est la racine de l'arbre que nous venons de décrire.

```

type T_Dico(Dico_Perso : Boolean := False) is
  record
    Nb_Mots      : Natural      := 0;
    Nb_Cel      : Positive     := 1;
    Nb_Car      : Natural      := 0;
    Racine      : T_Ptr_Noeud := new T_Noeud(False);
    case Dico_Perso is
      when True => Fichier : Ada.Strings.Unbounded.Unbounded_String;
      when False => null;
    end case;
  end record;

```

Les trois champs `Nb_Mots`, `Nb_Cel` et `Nb_Car` permettent de mémoriser respectivement le nombre de mots contenus dans le dictionnaire, le nombre de cellules constituant l'arbre et le nombre total de caractères qui composent tous les mots en mémoire.

Le discriminant `Dico_Perso` permet d'associer au dictionnaire un nom de fichier contenant le dictionnaire personnel associé au type `T_Dico`. Ainsi, il est possible de sauvegarder sur disque des mots que l'utilisateur ajoute au dictionnaire.

2.1.7 Les propositions

Cette dernière structure concerne la liste des mots proposés par le dictionnaire lorsqu'un mot soumis à son contrôle n'existe pas en mémoire. Pour cela, nous avons déclaré dans la partie public les types suivants :

```
-- Liste de mots contenant les propositions de correction
type T_Liste_Mots is private;

-- Nombre maximum de mots proposés par la procédure Controller
Propositions_Max : constant := 100;

-- Indice des mots proposés dans T_Liste_Mots
subtype T_Indice_Propositions is Integer range 1..Propositions_Max;
-- Nombre de mots proposés contenu dans T_Liste_Mots
subtype T_Nombre_Propositions is Integer range 0..Propositions_Max;
```

La première ligne déclare le type contenant les propositions retournées par la procédure `Controler`. Nous avons limité à 100 le nombre de propositions pouvant être retournées par notre type. Lors de tous nos tests, cette valeur n'a encore jamais été atteinte, et il nous a semblé qu'un dictionnaire qui propose plus de 100 mots par requête devenait vite inutilisable.

Les deux derniers sous-types sont utilisés dans les paramètres de la procédure `Controler`.

La déclaration complète du type `T_Liste_Mots` est composée d'un enregistrement contenant un tableau de chaînes de caractères non contraintes et d'un compteur permettant de mémoriser le nombre de mots présents :

```
-- Tableau de propositions
type T_Tab_Propositions is array(T_Indice_Propositions) of
  Ada.Strings.Unbounded.Unbounded_String;

-- Type contenant les propositions.
type T_Liste_Mots is
  record
    Nombre : T_Nombre_Propositions := 0;
    Mot    : T_Tab_Propositions;
  end record;
```

2.2 Architecture du programme

Une fois la structure de donnée clairement définie, on peut considérer qu'« il ne reste plus qu'à coder » ! Il reste en fait quelques choix à faire, notamment en ce qui concerne

l'architecture du programme. Nous avons axé l'application autour d'un paquetage `Correcteur_Orthographique` qui implémente un type de données abstrait dictionnaire : `T_Dico`.

La figure 5 montre comment nous avons organisé les fichiers qui composent le programme.

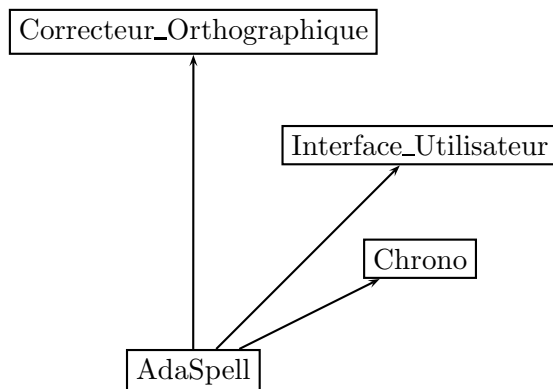


FIG. 5 – Graphe des appels entre les différents fichiers du programme.

Pour une meilleure lisibilité et réutilisabilité, les fonction principales du programme `AdaSpell`, soit les corrections interactives et automatiques, sont placées dans des procédures *Separate*.

2.2.1 AdaSpell

C'est le programme principal. Il déclare un dictionnaire courant et interprète les arguments passés en ligne de commande pour connaître les fichiers avec lesquels il doit travailler, ainsi que le mode de correction, automatique ou interactive.

2.2.2 Correcteur_Orthographique

Ce paquetage définit le type de données utilisé pour stocker le dictionnaire (voir page 5). Ce type est utilisé par le type privé `T_Dico`. L'utilisateur peut déclarer un dictionnaire et y appliquer les opérations suivantes :

- savoir s'il est vide ;
- connaître le nombre de mots qu'il contient ;
- y ajouter un mot ;
- charger un dictionnaire (y ajouter une liste de mots) ;

- y attacher un dictionnaire personnel ;
- savoir si un mot est dans le dictionnaire ;
- contrôler un mot pour éventuellement recevoir en retour une liste de propositions ;
- connaître le nombre de propositions faites ;
- retourner une proposition en particulier ;
- libérer la mémoire occupée par un dictionnaire ;

Les algorithmes d'insertion, de recherche et de contrôle d'un mot sont donnés en pseudo-code et dans les grandes lignes ci-dessous. Comme indiqué dans la spécification, les types de propositions sont au nombre de quatre (voir page 3).

Notons que la procédure qui affiche les propositions de correction est intelligente ; si la procédure de correction voit le mot `wal`, elle va à un moment ajouter un 'l' en troisième position et proposer le mot `wall`. Plus tard, elle va ajouter un 'l' en quatrième position et proposer de nouveau le mot `wall`. La liste des propositions ne contiendra qu'une seule fois le mot `wall`.

De même, les propositions respectent la casse des mots identifiés comme ne faisant pas partie du dictionnaire.

Algorithme 1 Insérer un mot dans le dictionnaire

```

déclarer un pointeur courant sur la racine du dictionnaire
pour chaque caractère du mot jusqu'à l'avant-dernier faire
    indiquer que la lettre courante existe
    si courant pointe sur une feuille alors
        créer un nouveau nœud
        échanger les valeurs de la feuille et du nœud
        faire pointer l'ancien nœud sur le nouveau et le nouveau sur la feuille
    sinon
        si la feuille suivante n'existe pas encore alors
            créer la feuille suivante et descendre le pointeur
        finsi
    sinon
        si la feuille suivante existe alors
            descendre le pointeur
        finsi
    finsi
fin pour

```

2.2.3 Interface_Utilisateur

Ce paquetage fait partie de nos composants réutilisables ; il permet de gérer menus, saisies contrôlées messages à l'attention de l'utilisateur.

Algorithme 2 Savoir si un mot se trouve dans le dictionnaire

déclarer un pointeur courant sur la racine du dictionnaire

convertir le mot en minuscules

pour chaque caractère du mot jusqu'à l'avant-dernier **faire**

si le caractère n'existe pas dans le tableau courant **alors**

 retourner FAUX, le mot n'est pas dans le dictionnaire

sinon

 (on sait que le caractère existe)

si on peut encore descendre d'un niveau **alors**

 descendre le pointeur d'un niveau

sinon

 retourner FAUX (on ne peut plus descendre dans l'arbre)

finsi

finsi

fin pour

retourner la valeur du bit qui, dans la feuille courante, indique si le dernier caractère du mot est terminal.

2.2.4 Chrono

Ce paquetage, développé par Pascal Pignard[3], nous permet d'indiquer avec précision le temps nécessaire au chargement du dictionnaire d'une part et à l'analyse d'un fichier d'autre part.

Algorithme 3 Contrôler un mot

si le mot n'est pas dans le dictionnaire **alors**

(proposer des mots avec une lettre en moins)

pour chaque lettre du mot **faire**

afficher le mot si il est présent sans cette lettre

fin pour

(proposer des mots avec une lettre en plus)

pour chaque interstice du mot (y. c. avant et après) **faire**

pour tous les caractères du type **faire**

insérer le caractère dans l'interstice

afficher le mot ainsi composé si il est présent dans le dictionnaire

fin pour

fin pour

(proposer des mots avec une deux lettres inversées)

copier le mot

pour la position i de chaque lettre du mot, jusqu'à l'avant dernière **faire**

inverser les lettres i et $i + 1$ dans la copie

afficher la copie si elle est dans le dictionnaire

fin pour

(proposer des mots avec une lettre différente)

pour la position de chaque lettre du mot **faire**

pour chaque lettre de l'alphabet à cette position **faire**

afficher le nouveau mot s'il est dans le dictionnaire

fin pour

fin pour

finsi

3 Guide utilisateur

3.1 Introduction

`AdaSpell` est un programme qui constitue un dictionnaire d'après une liste de mots. Il permet ensuite d'aider à corriger des textes en proposant des corrections aux mots qui ne sont pas dans le dictionnaire.

Le dictionnaire de `AdaSpell` ne contient que les 26 lettres de l'alphabet ; les autres caractères sont considérés comme des caractères de séparation. Aussi, `AdaSpell` est pour l'instant incapable de corriger le français.

3.2 Les fichiers

Dans cette présentation du programme, on suppose que le programme `AdaSpell` et les fichiers qui l'accompagnent sont organisés selon la hiérarchie suivante :

```
/adaspell
/dicos/
/dicos/dico.txt
/dicos/web2
/textes/
/textes/1984.txt
/textes/apple.txt
/textes/gnat_rm.txt
/textes/simple.txt
/textes/Tom_Sawyer.txt
```

Les fichiers dans le dossier `/dicos/` sont des listes de mots servant à élaborer les dictionnaires ; les fichiers dans le dossier `/textes/` sont là pour être corrigés.

3.3 Utilisation du programme

`AdaSpell` s'utilise en ligne de commande. Les dictionnaires à utiliser, les éventuelles options, le fichier à corriger et le fichier de destination sont passés en paramètre en ligne de commande.

L'utilisation typique du programme est donnée par sa notice, présentée à la figure 6.

Pendant le chargement d'un dictionnaire, le programme affiche une séquence de points indiquant le nombre de mots déjà chargés ; un point correspond à 10 000 mots. Une fois que le dictionnaire est chargé, le programme affiche le nombre de caractères et le nombres de mots contenus dans le dictionnaire. Il affiche également le nombre de pointeurs nécessaires à sa représentation en mémoire, ainsi que le temps de chargement en millisecondes.

```
AdaSpell 1.0 - (c) 2004 - D. Lifschitz, N. Seriot
Correcteur d'orthographe (caracteres standards seulement).

Usage :
adaspell -d:chaine {+ -d:chaine} [-p:chaine] [-auto] source cible
  -d:      le dictionnaire principal
  -p:      le dictionnaire personnel
  -auto    Enregistre les propositions sur le disque
  source   Fichier texte a controler
  cible    Fichier de sortie

Exemple:
  adaspell -d:dicos/dico.txt -p:perso.txt textes/apple.txt out.txt
```

FIG. 6 – Notice d'utilisation de AdaSpell

3.3.1 La correction en mode interactif

Par défaut, *AdaSpell* corrige les textes en mode interactif, c'est-à-dire que pour chaque erreur rencontrée, le programme permet à l'utilisateur de déterminer l'action à effectuer. Selon les cas et les éventuelles propositions de correction, l'utilisateur peut choisir la manière dont il faut traiter le mot, au moyen des touches numériques. Le traitements peut être :

- ignorer le mot ;
- choisir une proposition dans une liste ;
- afficher la prochaine page de propositions ;
- afficher la page précédente de propositions ;
- saisir un nouveau mot ;
- ajouter le mot au dictionnaire utilisateur.

3.3.2 La correction en mode automatique

Ce mode est activé par l'option `-auto`. Il permet, pour un fichier donné, d'identifier les mots qui ne sont pas dans le dictionnaire et de créer un nouveau fichier contenant, pour chaque mot, la liste des propositions. Ce mode indique le temps nécessaire à la correction du fichier.

4 Tests

Afin de tester le TDA `Correcteur_Orthographique`, nous avons écrit un programme de tests faisant appel à toutes ses procédures et fonctions exportées.

Le programme `Correcteur_Orthographique_T` livré avec les sources du projet effectue dans l'ordre les opérations suivantes :

- chargement en mémoire du dictionnaire *dico.txt* ;
- contrôle de la présence de tous les mots contenus dans *dico.txt* ;
- effacement du dictionnaire de la mémoire ;
- nouveau chargement du dictionnaire *dico.txt* et contrôle de la présence de tous les mots ;
- attachement d'un dictionnaire personnel ;
- recherche et affichage du mot *dictionary* mal orthographié :
 - *ditcionary*, soit inversion de deux lettres ;
 - *ictionary* et *dicionary*, soit une lettre en moins ;
 - *dictionnary*, soit une lettre en plus ;
 - *dycionary*, soit une lettre mal orthographiée.
- insertion du mot *eivd* dans le dictionnaire personnel ;
- recherche du mot *eivd*.

Remarque Avant de lancer le programme de test, il est nécessaire d'effacer le fichier *test_perso.txt* du répertoire courant. En effet, ce fichier correspond au dictionnaire personnel, et le mot *eivd* y est inséré.

Il faut aussi que les fichiers soient tels que présentés page 17.

5 Journal de travail

lundi 2 février

- Pris connaissance du sujet, premières recherches et discussions.

mardi 3 février

- Codage de deux prototypes : l'un utilisant le paquetage `Arbres_Avl` du cours et stoquant des `Unbounded_Strings`, l'autre utilisant un arbre 26-aire, insertion d'un dictionnaire conséquent, comparaisons de performance / occupation mémoire. L'insertion dans l'arbre `avl` de `Unbounded_Strings` est nettement plus lente.

mercredi 4 février

- Nouveau prototype à partir de l'arbre n-aire de la veille, en minimisant les déclarations de cellules à l'aide d'un article à discriminant.
- Optimisation de l'occupation mémoire à l'aide de pragmas, tests d'occupation mémoire et de performance. Le nouveau prototype économise de l'ordre de 20% de mémoire pour un dico de 240 000 mots.
- Implémentation de la recherche.

jeudi 5 février

- Discussion sur la mémoire réellement économisée grâce au discriminant. Il semblerait que le compilateur réserve la mémoire même pour les champs inutilisés.
- Réimplémentation à l'aide d'un tableau de cellules de un pointeur sur `T_Info` et un pointeur sur `T_Tab_Cel`. Surprise, on économise rien, on a toujours 127 Mo (avec `web2`) et sans `Pragma`. De plus, le `pragma Pack` ne sert plus à rien.
- Réimplémentation à l'aide d'un record avec les valeurs booléennes `Existe` et `Terminal` ainsi qu'un pointeur sur le tableau suivant. On peut packer efficacement `T_Cellule`. Meilleure implémentation jusque là (80 Mo).
- Codage des procédures affichant les propositions de corrections (4 cas).
- Codage de l'interprétation (parsing) d'un fichier d'entrée.
- Formattage des résultats dans un fichier de sortie.
- Mise en place d'une interface utilisateur minimale.

vendredi 6 février

- Amélioration générale du code.

samedi 7 février

- Corrigé un bug qui faisait planter le programme si le fichier à contrôler ne se terminait pas par un retour chariot.
- Ajouté un chronomètre.
- Ajouté un compteur d'erreurs.
- Gestion des exceptions.
- Améliorations générales.

- Mise en forme des commentaires.
- Tests...

dimanche 8 février

- Rédaction du rapport

lundi 9 février

- Modification du paquetage Correcteur_Orthographique dans le but d'en faire un « vrai »TDA ; il exporte maintenant un type T_Dico, en fait un enregistrement dont les champs contiennent ses attributs et un pointeur sur l'arbre des mots. Les procédures et fonctions du programme prennent un dictionnaire en paramètre.
- Ajouté la possibilité d'ajouter un mot au dictionnaire.
- Mis à jour le rapport.

mardi 10 février

- Discussion sur les objectifs du TDA Dico. Il en ressort qu'il faut pouvoir gérer un dictionnaire personnel qui contient les mots ajoutés par l'utilisateur. Ces mots doivent aussi pouvoir être sauvegardés dans un fichier. En outre, il faudrait que la procédure Contrôler soit capable de retourner une liste de proposition si le mot passé en paramètre n'existe pas dans le dico courant.
- Discussion sur le bien fondé d'avoir la procédure Controller_Fichier dans le TDA. Il faudra probablement le sortir.
- Codage de la procédure qui efface le dictionnaire de la mémoire.
- Ajout d'un type privé T_Liste_Mots permettant de retourner une liste de propositions pour les mots absents du dictionnaire.
- Adaptation de la procédure Contrôler afin qu'elle retourne une liste de propositions en lieu et place de l'affichage par des Put_Line.
- Modification de la procédure Insérer. Elle s'appelle désormais Ajouter et prend un paramètre de plus, à savoir le nom d'un fichier de sauvegarde.
- Ajouté les procédures Nombre_Proposition et Proposition qui permettent de manipuler la liste des mots contenant des propositions.

mercredi 11 février

- Adapté la procédure interne Ajouter proposition afin qu'elle n'ajoute pas de mots à double.
- Adapté la procédure Controller_Fichier afin de tenir compte des modifications de la procédure Contrôler.
- Mise à jour du rapport.

jeudi 12 février

- Codage d'un prototype de programme en ligne de commande permettant de contrôler un fichier texte à la volée.

samedi 14 février

- Ajout de la procédure Attacher_Dictionnaire_Perso permettant d'associer un tel dictionnaire au type T_Dico.
- Modification du type T_Dico pour mémoriser le dictionnaire personnel.
- Modification des procédures Vider et Ajouter afin de tenir compte du dictionnaire personnel.
- Mise à jour du rapport.

dimanche 15 février

- Mis la procédure Controller_Fichier dans un paquetage enfant, changé le nom en Controler_Fichier (avec un seul 'L'!)
- Modifié l'implémentation de la procédure Controler; les propositions respectent désormais la casse du mot examiné.
- Écrit un prototype permettant de corriger un fichier en ligne de commande, ceci de manière interactive : Check.adb.

mardi 17 février

- Réimplémenté le type de données pour ne pas déclarer inutilement des tableaux de 26 pointeurs sur les feuilles de l'arbre. Pour l'instant, cela se fait par, sur chaque nœud, un pointeur sur un tableau de 26 pointeurs. On économise effectivement de la mémoire, mais on déclare un pointeur en trop dans les nœuds qui ne sont pas des feuilles...

mercredi 18 février

- Réimplémenté le type de données de la veille avec un type article à discriminant sans valeur par défaut. On gagne encore plus de mémoire, mais l'insertion devient très longue et la recherche est aussi plus longue, malgré quelques améliorations à la fonction Est_Present.
- Regroupé toutes les versions en un dossier ancien (jusqu'au 16 février) et un dossier nouveau, permettant d'expérimenter les améliorations à notre structure de données.
- Rassemblé et synthétisé les différentes mesures effectuées (mesure du temps avec le paquetage Chrono et rapports de gprof).

vendredi 20 février

- Entériné le choix de la dernière structure expérimentés (article à discriminant sans valeur par défaut). Mis à jour le rapport, expliqué le cheminement qui nous a conduit à cette structure, fait état de certaines mesures.

dimanche 22 février

- Corrigé une erreur de version. Terminé de fusionner les progrès réalisés sur les fichiers associés à l'ancienne structure avec les fichiers associés à la nouvelle structure.

- Identifié un bug : avec check.adb : le caractère '>' n'est pas copié dans le nouveau fichier quand il se trouve en fin de ligne (voir apple.txt).
- Il reste à recoder la procédure qui vide le dictionnaire.

mardi 24 février

- Codage de la procédure vider.
- Identifié et corrigé un bug : les directives de compilation Pragma Pack provoquent des erreurs lors de la copie des éléments d'un nœud dans un autre.
- Fusionné labo.adb et check.adb en adaspell.adb.
- Mis à jour le rapport.
- Écrit le guide utilisateur.

mercredi 25 février

- Codage du programme de test pour le TDA.
- Mise à jour du rapport.
- Corrigé aide dans adaspell.
- Corrections et précisions générales dans le rapport.
- Ébauche de conclusion.

jeudi 26 février

- Corrigé le bug du caractère '>' identifié le 22 février.
- Ajouté l'exception `Erreur_Insertion` qui peut être levée par la procédure `Ajouter` si le mot passé en paramètre contient des caractères interdits.
- Ajouté l'exception `Erreur_Recherche` à la procédure `Controler` pour les mêmes motifs.
- Traité l'exception `Erreur_Insertion` dans `AdaSpell` si l'utilisateur saisit un nouveau mot contenant des caractères interdits.
- Corrections de style dans les sources.
- Corrigé un bug dans la détection du nombre de paramètres passés sur la ligne de commande dans `AdaSpell`.
- Corrections du rapport.

dimanche 29 février

- Corrections mineures du rapport

6 Conclusion

Nous avons rempli les objectifs que nous nous étions fixés. Nous avons déterminé et analysé un type de données abstrait qui correspond à nos besoins et réalisé un programme qui l'utilise. Le programme fonctionne bien, de manière efficace, tout en offrant deux modes de correction.

Le choix de la structure de données était clairement un des points les plus subtils de ce projet. La théorie sur les arbres que nous avons étudié au cours nous a permis de bien comprendre la problématique et d'imaginer puis de coder une structure adaptée.

Cependant, comme c'est souvent le cas en informatique, nous arrivons à un point où il s'agit de trouver un équilibre entre la performance et l'occupation de la mémoire. Bien que les directives de compilation permettent de favoriser l'un ou l'autre de ces aspects, le choix des structures de données et des algorithmes les manipulant sont, en définitive, les deux facteurs primordiaux.

En ce qui concerne le développement, nous avons tenté de suivre un *modèle en spirale*, qui consiste à toujours maintenir une version à jour, *livrable*, contrairement aux autres laboratoires où nous avons plutôt l'habitude d'un développement *en cascade*, où nous codions une partie après l'autre.

Ce modèle en spirale nous a aidé pour ce qui concerne la structure de données ; nous avons pu, par raffinements successifs, nous approcher d'une structure qui nous convenait. Par contre, les évolutions successives sont difficilement compatibles avec la nécessité de conserver une documentation à jour ; il n'est pas toujours facile de supprimer une documentation qui fût longue et difficile à rédiger, mais qui est devenue obsolète. Nous avons perdu beaucoup de temps dans cette activité de maintenance de la documentation.

Références

- [1] Projet EPFL. http://lglwww.epfl.ch/teaching/programming01_02/.
- [2] The Big Online Book of Linux Ada Programming. <http://www.vaxxine.com/pegasoft/homes/book.html>.
- [3] Blady : Programmation et Algorithmes. <http://perso.wanadoo.fr/blady/>.
- [4] *Webster's Second International*. Ce dictionnaire contient près de 235 000 mots. Son copyright, qui datait de 1934, est arrivé à expiration. Il est fourni avec la plupart des distributions de UNIX. Par exemple, sous Mac OS X/Darwin, il se trouve dans `/usr/share/dict/`.
- [5] Abdelali Guerid, Pierre Breguet, Henri Röthlisberger. *Algorithmes et structures de données avec Ada, C++ et Java*. Presses polytechniques universitaires romandes, 2002.