Licence d'informatique
Module de C/Unix

Génération automatique de textes

Examen – Parties Unix et C

Philippe Marquet

Janvier 2000

Les documents de cours et TD sont autorisés.

Ce sujet propose de développer une solution en deux parties : une partie écrite en langage C et une partie écrite en Bourne-shell.
On rendra deux copies séparées pour la partie langage C et pour la partie Unix.
La partie C (questions 1 à 4) est à rendre sur une *copie blanche* ; la partie Unix (questions 5 à 7) sur une *copie de couleur.*

Lire le sujet jusqu'au bout avant de répondre[0]. En particulier vous pouvez utiliser les outils présentés dans l'annexe.

La commande `ecrivain` produit sur sa sortie standard un texte en français qui se lit bien. Afin de générer un tel texte, la commande intègre un modèle statistique de la langue française. Il est « difficile » de synthétiser la langue française dans un modèle ; aussi la commande `ecrivain` va, dans un premier temps, utiliser un texte existant pour construire un modèle statistique de la langue utilisée dans ce texte et, dans un second temps, générer un texte aléatoire à partir de ce modèle.

Algorithme des chaînes de Markov

L'algorithme des chaînes de Markov permet de traiter ce type de problèmes. L'entrée est vue comme une séquence de phrases se recouvrant. Une phrase est composée de deux parties : un préfixe de plusieurs mots et un suffixe, le mot unique qui suit le préfixe. L'algorithme produit un texte en choisissant aléatoirement un suffixe parmi ceux associés au préfixe dans le texte original.

Dans notre cas, un préfixe de deux mots produit un résultat correct[1]. L'algorithme peut être décrit de la manière suivante :

```
soient w_1 et w_2 les deux premiers mots du texte ;
afficher w_1 et w_2 ;
forever do
    choisir aléatoirement w_3 parmi les succes-
seurs du préfixe w_1 w_2 ;
    afficher w_3 ;
    remplacer w_1 et w_2 par w_2 et w_3 ;
done
```

[0]La lecture des notes de bas de pages telles celle-ci n'est cependant pas indispensable pour traiter le sujet.
[1]Réduire la taille du préfixe tend à produire des textes moins cohérents, augmenter cette taille tend à reproduire plus/trop exactement le texte d'entrée.

On définit un mot de la manière suivante. Un mot est une suite de caractères alphabétiques. Les mots sont séparés par des espaces. On laisse la ponctuation attachée aux mots : « *mot* » et « *mot.* » sont des mots différents[2].

Exemples

Soit le « texte » suivant :

`aa bb` <u>`cc`</u> `dd ee aa bb` <u>`dd`</u> `ee cc dd aa bb` <u>`ee`</u> (*fin*)

Les phrases de ce texte sont les suivantes (étudiez par exemple les apparitions du préfixe `aa bb` dont les suffixes sont soulignés dans le texte) :

Préfixe :	Suffixes possibles :
`aa bb`	<u>`cc`</u>, <u>`dd`</u>, <u>`ee`</u>
`bb cc`	`dd`
`cc dd`	`ee`, `aa`
`dd ee`	`aa`, `cc`
`ee aa`	`bb`
`bb dd`	`ee`
`ee cc`	`dd`
`bb ee`	(*fin*)

Le texte généré débutera par `aa bb`, l'algorithme choisira ensuite un des suffixes parmi `cc`, `dd`, et `ee`. Si `dd` est choisi, le préfixe courant devient `bb dd` et on choisira ensuite nécessairement `ee`. Le préfixe courant est alors `dd ee` et on pourra choisir `aa` ou `cc`. La génération se poursuit ainsi jusqu'à ce que le texte produit soit assez long ou que le marqueur de fin ait été généré.

Le texte suivant a été généré à partir d'anciens sujets d'examen :

> La commande Bourne-shell entry qui affiche le nom est passé en paramètre sous la forme du code Bourne-shell qui, à partir d'une arborescence originelle et d'un fichier non défini comme une nouvelle version. Les options -C et -D spécifient d'utiliser les commandes précédentes ont été positionnées en conséquence. Donnez le code Bourne Shell de l'algorithme proposé à la question ont été appliquées. Vous ferez figurer les partitions. Donnez une liste de variables d'environnement de modifier la configuration de MC. Il définit entre autres la variable PATH est gênante. Expliquer en quoi il est intéressant que les répertoires.

Le texte suivant a été généré à partir de *Bouvard et Pécuchet* de Gustave Flaubert.

> De temps à autre, il lâchait un juron puis, sur un tabouret. Pendant dix minutes, ils demeurèrent dans cette intolérable odeur, une odeur féroce et comme ils parlaient des menhirs et des bandes traînaient sur la manière d'une lanterne magique. Ses personnages, alertes comme des pains de sucre, tenus en l'air – pas davantage ! Celle de Victorine était généralement unie, marque de décadence.

Architecture de nos développements... donc de l'examen

Nous allons développer une commande `ecrivain` en C. Afin de tester cette commande, nous développerons en Bourne-shell une commande `test-ecrivain`.

[2] Ainsi le texte généré sera ponctué. Cependant, cette approche peut produire des textes mal ponctués : parenthèse non concordante, guillemets incorrects, etc.

1 Partie C de l'examen

La commande `ecrivain`

La syntaxe de la commande `ecrivain` est la suivante :

`ecrivain [-p psize] [-s size] [inputfile [outputfile]]`

Elle génère dans le fichier *outputfile* (par défaut la sortie standard) un fichier d'au plus *size* mots (par défaut 1000), à partir du fichier *inputfile* (par défaut l'entrée standard). L'algorithme utilise des préfixes de *psize* mots (2 par défaut).

Une structure de données pour la commande `ecrivain`

Pour que l'algorithme de génération de texte fonctionne correctement, le volume de l'entrée doit être relativement important, par exemple le contenu complet d'un livre[3].

Nous allons construire une structure de données qui associe un préfixe et l'ensemble de ses suffixes. Lors de la première phase de l'algorithme (construction de la structure de données à partir du texte d'entrée), il est nécessaire de retrouver la structure d'un préfixe donné pour y ajouter un nouveau suffixe. Lors de la seconde phase de l'algorithme (génération du texte), il est nécessaire de retrouver la structure d'un préfixe donné pour choisir aléatoirement un de ses suffixes. Dans les deux cas, l'algorithme demande de retrouver rapidement l'ensemble des suffixes d'un préfixe donné. Nous pouvons utiliser une table de hachage.

Le principe d'une table de hachage est de partitionner l'ensemble des éléments en un nombre N, fixé, de listes. Une table de hachage est ainsi vue comme un tableau de N listes d'éléments. Une fonction (la fonction de hachage) retourne pour un élément donné un entier qui est un indice dans ce tableau. Ainsi, la recherche de la position d'un élément dans la table de hachage[4] commence par calculer cet indice ; avec cet indice, on accède dans la table de hachage à une liste d'éléments ; on recherche alors séquentiellement dans la liste correspondante l'emplacement de l'élément. L'efficacité d'une table de hachage réside dans le choix judicieux de la fonction de hachage ; son exécution doit être peu coûteuse et les indices produits par cette fonction sur l'ensemble des éléments doivent bien se répartir entre 0 et $N-1$.

La structure de données associée à un préfixe est la suivante :

```
unsigned int PLEN ;     /* longueur des préfixes (nombre de mots dans un préfixe)

typedef struct prefix Prefix ;
typedef struct suffix Suffix ;

struct prefix {
  char **words ;        /* le tableau (de taille PLEN) des mots du préfixe */
  Suffix *suffixes ;    /* la liste des suffixes */
} ;

struct suffix {
  char *word ;          /* la valeur du suffixe */
  Suffix *nextS ;       /* le suivant dans la liste des suf-
fixes du préfixe */
} ;
```

[3] On s'apprête donc à traiter de l'ordre de 100.000 mots en entrée et de l'ordre de 1.000 mots en sortie. La commande ne peut reposer sur une structure de données simpliste (par exemple la liste des mots) sous peine de performances détestables : 1000 fois une recherche séquentielle dans une liste de 100.000 mots.

[4] que ce soit pour accéder à l'élément ou l'insérer dans la table

La table de hachage est alors déclarée ainsi :

```
#define NHASH 4093      /* taille du tableau consti-
tuant la table de hachage */

/* Un élément d'une liste de la table de hachage */
typedef struct elem Elem ;
struct elem {
  Prefix prefix ;       /* le prefixe */
  Elem *nextE ;         /* l'élément suivant dans la liste */
} ;

/* La table de hachage */
Elem *hashtable [NHASH] ;
```

La figure 1 illustre cette structure de données.

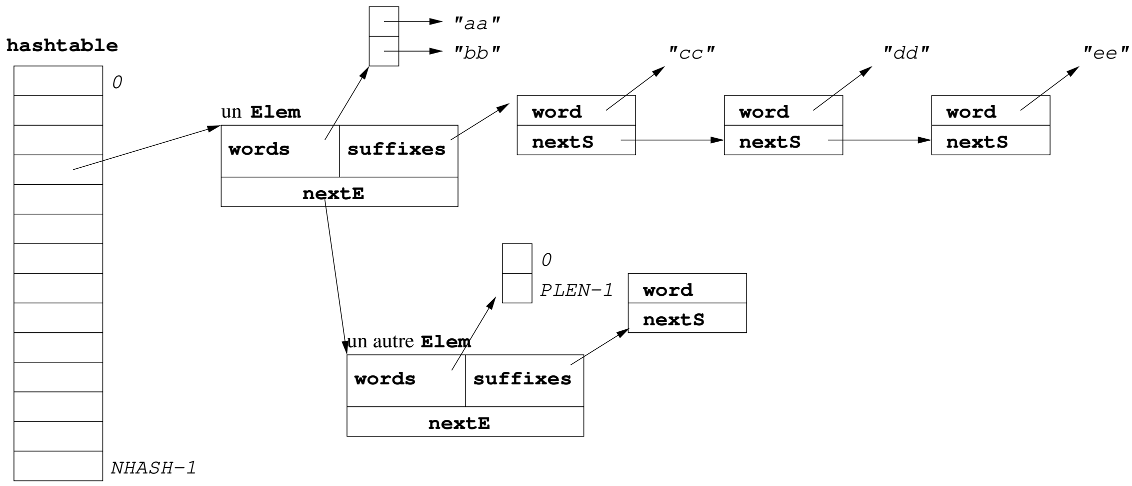

FIG. 1 – Une table de hachage pour `ecrivain` (`PLEN` vaut 2)

On utilise la fonction de hashage suivante :

```
/* la fonction de hashage. Retourne un entier inférieur à NHASH à
   partir d'un tableau de PLEN mots constituant un préfixe. */
unsigned int hash (char *words[]) ;
```

Questions – Partie C

Question 1

La table de hachage est organisée sous forme de listes de préfixes lesquels sont mémorisés sous forme de tableaux de `PLEN` mots. Nous avons besoin d'une fonction de test d'égalité de deux préfixes :

```
int prefixeql (const char *words1[], const char *words2[]) ;
```

retourne zéro si et seulement si les deux préfixes sont différents.

Donnez le code C de la fonction `prefixeql ()`.

Question 2

La fonction `addsuffix ()` ajoute un suffixe donné à la liste des suffixes d'un préfixe donné :

```
void addsuffix (Prefix *p, const char *suffix) ;
```

Il est garanti que la chaîne de caractères `suffix` sera inchangée dans la suite du programme (elle n'a donc pas à être recopiée).

On ne se préoccupe pas de savoir si le suffixe est déja présent dans la liste des suffixes[5].

Donner le code C de la fonction `addsuffix ()`.

Question 3
La fonction `lookup ()` recherche un préfixe (tableau `words` de `PLEN` chaînes de caractères) donné dans la table de hachage. Elle retourne un pointeur sur le préfixe.

```
Prefix *lookup (const char *words[], unsigned int create) ;
```

Le paramètre `create` indique le comportement de `lookup ()` dans le cas où le préfixe n'est pas présent :
– si `create` est nul, `lookup ()` retourne `NULL;`
– sinon, `lookup ()` ajoute le préfixe à la table de hachage `hashtable`.
  Dans ce cas, il est garanti que les chaînes de caractères de `words` seront inchangées dans la suite du programme. Inversement, le tableau `words` doit être dupliqué.

Donner le code C de la fonction `lookup ()`.

Question 4
La fonction `getasuffix ()` retourne un suffixe choisi aléatoirement parmi les suffixes du préfixe donné par le tableau de `PLEN` chaînes de caractères :

```
char *getasuffix (char *words[]) ;
```

Il est garanti qu'un tel suffixe existe dans la table de hachage[6].

2 Partie Unix de l'examen

Important Plusieurs commandes/fonctions Bourne-shell sont à développer. On veillera particulièrement à détailler pour chacune des commandes un algorithme avant de donner le code Bourne-shell lui-même. La notation sera répartie entre ces deux composantes de vos réponses.

Test de la commande `ecrivain`

Comment tester qu'un programme écrit pour la commande `ecrivain` ci-dessus est valide ? Cette commande produit du non-sens qu'il est difficile d'analyser. De plus, le résultat est aléatoire et différent à chaque fois.

Un premier ensemble de tests peut vérifier que pour les cinq textes en entrée suivants (un texte par ligne), le résultat est identique à l'entrée :

(*fichier vide*)
```
a
a b
a b c
a b c d
```

Un second ensemble de tests peut vérifier la propriété de conservation de l'algorithme. Pour des préfixes de longueur 2, tous les mots, couples et triplets du résultat doivent appartenir au texte original.

[5] Plus un suffixe apparaît dans le texte original, plus il doit avoir de chance d'apparaître dans le texte généré. Nous ne gérons pas un compteur associé à chacun des suffixes mais procédons par présence multiple du suffixe dans la liste des suffixes.

[6] On sort du programme sur une erreur fatale (un bug de notre programme !) sinon. En effet pour tout préfixe du texte initial, il y a un suffixe dans la table de hachage. Seul le dernier préfixe du texte pourrait ne pas être présent dans la table. On a donc ajouté, en fin de création de la table, le suffixe `""` associé à ce dernier préfixe.

Un troisième ensemble de tests peut vérifier les propriétés statistiques du résultat. Par exemple soit l'entrée :

`a b c a b c` … `a b c a b d a b c` …

telle que le nombre de `a b c` est dix fois supérieur au nombre de `a b d`. Cette propriété statistique de l'entrée doit se retrouver dans le résultat produit.

La commande `test-ecrivain`

Nous allons développer une commande `test-ecrivain` pour exécuter ces tests. Sa syntaxe est la suivante :

`test-ecrivain` [`-l` *`level`*] [*`infile`*]

le paramètre *`level`* correspond à la série de tests que nous voulons effectuer parmi les trois familles décrites ci-dessus (valeur possible : 1 à 3 caractères parmi `1`, `2`, `3`; valeur par défaut : `123`). Pour le niveau 2 de test, le fichier *`infile`* est utilisé en entrée de la commande `ecrivain`.

Notre programme shell sera constitué des trois fonctions shell `test1`, `test2`, et `test3` qui réalisent les tests du niveau correspondant. Seule la fonction `test2` accepte un paramètre qui est le nom du fichier qui servira d'entrée pour la commande `ecrivain`.

Questions – partie Unix

Question 5
La première série de tests demande de vérifier l'égalité de deux fichiers (le fichier original et le fichier produit par `ecrivain`). Cette égalité s'entend au espaces et retour à la ligne près. Pour ce faire, nous développons une fonction Bourne-shell `normalize` qui produit sur la sortie standard le texte lu sur l'entrée standard à raison d'un mot par ligne, sans espace ni tabulation.

Question 5.1 Développez la fonction Bourne-shell `normalize`.

Question 5.2 Développez la fonction Bourne-shell `test1` qui travaille sur les cinq textes donnés en exemple.

Question 6
La mise en œuvre du second test nécessite d'identifier l'ensemble des triplets de mots d'un texte donné. Ce sera le rôle de la fonction Bourne-shell `triple`. Elle produit sur la sortie standard l'ensemble des triplets de mots du texte lu sur l'entrée standard. Ces triplets sont produits à raison d'un par ligne. Sur une ligne, les mots sont séparés par un espace unique.

Question 6.1 Développez la fonction Bourne-shell `triple`.

Question 6.2 Développez la fonction Bourne-shell `test2`.

Question 7
Pour les besoins du troisième test, nous utilisons une fonction Bourne-shell `generate` qui génère **aléatoirement** un texte composé de `a b c` et de `a b d` tel que le nombre de `a b c` est dix fois supérieur au nombre de `a b d`.

Question 7.1 Développez la fonction `generate`. On fixe arbitrairement la longueur du texte à 3000 mots (c'est-à-dire 1000 triplets `a b c` ou `a b d`).

Question 7.2 Développez la fonction Bourne-shell `test3` qui vérifie que le résultat produit par `ecrivain` sur le texte de `generate`[7] comporte entre 88% et 92% de `a b c` pour 8% à 12% de `a b d`.

[7] Remarquez que ce texte ne comporte que des `a b c` et des `a b d`. Compter les `a b c` revient donc à compter le `c`; de même pour les `a b d` et `d`.

3 Annexe

- La commande Unix `random` max[8] affiche sur sa sortie standard un entier aléatoire r tel $0 \leq r < max$. Cette valeur r est aussi celle retournée par la commande.
- La fonction

```
#include <stdlib.h>
long int random(void) ;
```

retourne un nombre aléatoire compris entre 0 et `RAND_MAX`. On peut considérer que

```
random () % n == 0
```

est vrai avec une probabilité de $\frac{1}{n}$.

[8]développée pour les besoins du sujet