# TP Algorithmique expérimentale (partie 1)

## Cas d'étude : un générateur automatique de texte

L'objectif est de produire automatiquement un texte aléatoire qui ressemble à un texte écrit par un être humain. Ce problème se pose dans différents contextes : parodie, évitement de filtres anti-spam… Une approche possible consiste à utiliser une méthode MCMC (Markov Chain Monte Carlo). L'idée est de partir d'un texte réel pour créer un espace d'états et des probabilités de transition entre états, puis de générer un « faux » texte en passant d'un état à l'autre en fonction de ces probabilités.

L'algorithme a besoin de trois entrées :

- un vrai texte *T* contenant *n* mots,
- le nombre *m* de mots que doit contenir le « faux » texte à générer,
- un entier *k* qui détermine la taille (en nombre de mots) des séquences-clés (voir plus bas).

Dans le texte *T* de départ, chaque séquence de k mots consécutifs est appelée clé. Une clé forme un préfixe, et le mot qui suit immédiatement la clé dans le texte est appelé suffixe. Par exemple, pour $k = 2$, voici toutes les clés et tous les suffixes du texte *T*= « this is a test this is only a test this is a test of the emergency broadcasting system » :

| Clé (préfixe) | Suffixe | Index du début de la clé dans le texte T |
|---|---|---|
| this is | a | 0 |
| is a | test | 5 |
| a test | this | 8 |
| test this | is | 10 |
| this is | only | 15 |
| is only | a | 20 |
| only a | test | 23 |
| a test | this | 28 |
| test this | is | 30 |
| this is | a | 35 |
| is a | test | 40 |
| a test | of | 43 |
| test of | the | 45 |
| of the | emergency | 50 |
| the emergency | broadcasting | 53 |
| emergency broadcasting | system | 57 |
| broadcasting system | this | 67 |
| system this | is | 80 |

Ce tableau est ensuite trié par ordre alphabétique des clés, pour regrouper les clés identiques :

| Clé (préfixe) | Suffixe | Index du début de la clé dans le texte T |
|---|---|---|
| a test | this | 8 |
| a test | this | 28 |
| a test | of | 43 |
| broadcasting system | this | 67 |
| emergency broadcasting | system | 57 |
| is a | test | 5 |
| is a | test | 40 |
| is only | a | 20 |
| of the | emergency | 50 |
| only a | test | 23 |
| system this | is | 80 |
| test of | the | 45 |
| test this | is | 10 |
| test this | is | 30 |
| the emergency | broadcasting | 53 |

| | | |
|---|---|---|
| this is | a | 0 |
| this is | only | 15 |
| this is | a | 35 |

Une fois ce tableau généré, le « faux » texte est initialisé avec les k premiers mots du vrai texte. Ces mots sont affichés et stockés dans une variable S de type chaîne de caractères. Puis les m–k mots restants sont générés de la façon suivante :

- recherche de toutes les clés qui correspondent à S,
- choix aléatoire d'une de ces clés, affichage du suffixe correspondant, mise à jour de S en supprimant son premier mot et en ajoutant à la fin le suffixe que l'on vient de choisir.

Par exemple, si *S* contient « this is », l'étape de recherche renverra trois suffixes possibles : « a », « only » et « a ». On a alors 2 chances sur 3 que le suffixe choisi soit « a » (dans ce cas, la nouvelle valeur de *S* sera « is a »), et 1 chance sur 3 que le suffixe choisi soit « only » (et dans ce cas, la nouvelle valeur de *S* sera « is only »).

Bien sûr, la valeur du paramètre *k* influence beaucoup l'allure du texte généré. Avec $k = 1$, la grammaire est complètement incorrecte, alors qu'avec $k = 3$, la grammaire est meilleure (bien qu'imparfaite).

## Si vous travaillez sous Linux

Créez un répertoire pour le TP et placez-y les les fichiers markov.c, zadig.txt, madame-bovary.txt, don-quixote.txt que vous trouverez à l'adresse :

http://liris.cnrs.fr/~cknibbe/connaissance-metier-recherche

Ouvrez un terminal et placez-vous dans le répertoire du TP.

### 1. Mesurer le temps d'exécution

Compilez le programme en mode debug :

```
gcc -g -o markov markov.c
```

Exécutez-le avec k = 3 et m = 1000000 :

```
./markov 3 1000000 <don-quixote.txt >resultat.txt
```

Mesurez le temps d'exécution et notez les 3 valeurs obtenues :

```
time ./markov 3 1000000 <don-quixote.txt >resultat.txt
```

Recommencez plusieurs fois en notant à chaque fois les valeurs obtenues. D'où peuvent venir les variations observées ?

Commentez l'appel à srand, recompilez et mesurez à nouveau plusieurs fois le temps d'exécution. Y a-t-il encore de la variabilité ?

Décommentez à présent l'appel à srand, recompilez, puis testez l'influence de la charge totale du système en lançant le programme en même temps que plusieurs autres applications. Le temps CPU est-il réellement indépendant de la charge du système ?

Recompiler avec l'option d'optimisation –O3 :

```
gcc -O3 -o markov markov.c
```

Et mesurez à nouveau le temps d'exécution. Quel pourcentage d'amélioration obtenez-vous comparé à une compilation sans optimisation ?

### 2. Trouver l'opération dominante

Recompiler avec –pg (attention, ne pas utiliser cette option en même temps que les options d'optimisation –O1, -O2, -O3 etc.) :

```
gcc -pg -o markov markov.c
```

L'exécutable produira un fichier gmon.out à chaque exécution, ce fichier gmon.out contient les informations de profiling qui peuvent ensuite être analysées par gprof :

```
./markov 3 1000000 <don-quixote.txt >resultat.txt

gprof markov gmon.out
```

Quelle est l'opération dominante, c'est-à-dire celle qui prend le plus de temps d'exécution ?

### 3. Compter le nombre d'appels de l'opération dominante

Dans cet algorithme, la fonction wordncmp est utilisée dans trois buts :
- initialisation du tableau word
- recherche dichotomique des préfixes candidats dans le tableau
- choix aléatoire d'un préfixe parmi les préfixes candidats

Nous allons compter le nombre d'appels à cette fonction dans les trois phases. Il faut pour cela éditer légèrement le code. Définissez trois variables count1, count2, count3 et incrémentez-les quand il le faut. Affichez leurs valeurs à la fin du programme. Affichez également la somme des 3.

Rappel : si x est de type unsigned long, alors on l'affichera dans la sortie d'erreur de la façon suivante : fprintf(stderr, « x vaut %lu\n », x) ;

Voir la page http://en.wikipedia.org/wiki/Standard_streams si vous ne connaissez pas la notion de sortie d'erreur.

## 4. Mesurer la complexité de l'algorithme en fonction de n, m et k

On veut tester l'influence de n, m et k sur la valeur de count3. Concevez un plan d'expérience, écrivez-le sur un « cahier de laboratoire » (papier ou numérique). Faîtes les tests et sauvegardez les résultats dans un tableur.

Indication : si vous souhaitez utiliser d'autres fichiers de départ, vous pouvez télécharger des œuvres au format txt sur le site du projet Gutenberg. Enlevez l'entête Gutenberg au début du fichier et la licence Gutenberg à la fin du fichier.

## Si vous travaillez sous Mac

Vous pouvez suivre quasiment les mêmes étapes que sous Linux. Si gcc n'est pas installé, installez XCode depuis l'App Store, puis démarrez XCode. Allez dans les Préférences / onglet Downloads, et demandez l'installation des 'Command-line tools'. Une fois que l'installation est faite, vous pouvez quitter XCode et relancer le Terminal, gcc devrait fonctionner.

Si gprof ne fonctionne pas (résultats vides), c'est vraisemblablement dû à un bug connu de Lion et peut-être aussi de Mountain Lion. Vous pouvez dans ce cas utiliser l'application « Instruments » (/Developer/Applications/Instruments.app). Au lancement, choisissez le template « Time Profiler ». Lancez l'exécution de votre programme dans le Terminal, suspendez-le immédiatement en tapant CTRL+z dans le terminal. Retournez alors dans Instruments et choisissez votre processus à l'aide du bouton « Choose Target / Attach to process ». Appuyez sur le bouton « Record ». Retournez dans le Terminal et remettez votre processus en marche en tapant « fg ». Après une seconde environ, vous pouvez arrêter l'enregistrement dans Instruments et examiner les résultats.

## Si vous travaillez sous Windows

Créez un répertoire pour le TP et placez-y les fichiers markov-windows.c, zadig.txt, madame-bovary.txt, don-quixote.txt que vous trouverez à l'adresse :

http://liris.cnrs.fr/~cknibbe/connaissance-metier-recherche

### 1. Installation de Code::Blocks

Si vous n'avez pas déjà installé de compilateur C/C++ (comme Visual C++ par exemple), vous allez devoir en installer un. Je vous conseille Code::Blocks, qui est gratuit, et téléchargeable à l'adresse :

http://www.codeblocks.org/downloads/26#windows

Lors de l'installation, choisissez une installation « Custom » et cochez, dans la catégorie « Contrib plugins », le plugin Profiler.

### 2. Compilation du programme

Dans Code::Blocks, allez dans le menu File / New Project et choisissez Console Application. Sélectionnez le langage C. Appelez votre projet Markov. Sélectionnez le compilateur GCC et choisissez de créer deux configurations (Debug et Release) comme suggéré par défaut.

Dans le volet de gauche, cliquez sur le fichier main.c. C'est un squelette de code que Code::Blocks a créé par défaut dans le projet. Remplacez le contenu de ce fichier par le contenu du fichier markov-windows que vous avez pris sur le site de l'UE.

Allez ensuite dans le menu Project / Build options. Cliquez sur Debug dans la liste des configurations à gauche de la fenêtre qui s'ouvre. Cochez l'option « Produce debugging symbols » (-g). Cliquez ensuite sur Release dans la liste des configurations, et cochez l'option « Optimize fully for speed (-O3) » (l'option -g doit être décochée).

Dans la barre d'outils en haut, choisissez la cible « Debug » du menu déroulant « Build target ». Allez dans le menu Build / Build. Cela compile le programme. Un nouveau fichier nommé Markov.exe doit avoir été créé dans le sous-dossier Markov\bin\Debug de votre dossier de TP.

### 3. Mesurer le temps d'exécution du programme

Tapez cmd dans la zone de recherche du menu Démarrer, cela lance l'invite de commandes. Allez dans votre répertoire de TP avec la commande cd. Puis lancez le programme avec la commande :

.\Markov\bin\Debug\Markov.exe 3 1000000 < don-quixote.txt > resultat.txt

Notez le temps d'exécution du programme qui a dû s'afficher. Vérifiez qu'un nouveau fichier a été créé : resultat.txt, qui contient le « faux » texte.

Recommencez plusieurs fois en notant à chaque fois les valeurs obtenues. D'où peuvent venir les variations observées ?

Commentez l'appel à srand, recompilez et et mesurez à nouveau plusieurs fois le temps d'exécution. Y a-t-il encore de la variabilité ?

Décommentez l'appel à srand et recompilez à nouveau. Testez maintenant l'influence de la charge totale du système en lançant le programme en même temps que plusieurs autres applications. Le temps CPU est-il réellement indépendant de la charge du système ?

Retournez dans Code ::Blocks et choisissez cette fois la cible Release dans le menu déroulant « Build target » de la barre d'outils. Recompiler en allant dans le menu Build/Build. Mesurez le temps d'exécution du nouvel exécutable dans l'invite de commande :

.\Markov\bin**Release**\Markov.exe 3 1000000 < don-quixote.txt > resultat.txt

Quel pourcentage d'amélioration obtenez-vous comparé à la compilation précédente (sans optimisation et avec profiling) ?

### 5. Trouver l'opération dominante

Dans Code::Blocks, allez dans le menu Project / Build options et choisissez la configuration Debug à gauche. Cochez l'option « Profile code when executed » (-pg) et recompilez en choisissant bien la cible Debug.

L'exécutable produit ainsi produira un fichier gmon.out à chaque exécution, ce fichier gmon.out contient les informations de profiling qui peuvent ensuite être analysées avec le plugin Profiler de Code::Blocks. Allez donc dans l'invite de commande pour réexécutez le programme (prenez bien la version Debug), puis revenez dans Code::Blocks. Allez dans le menu Plugins / Code profiler. Une fenêtre vous indique que le fichier gmon.out ne se trouve pas dans le dossier courant (c'est normal, il n'est pas dans le même dossier que l'exécutable). Indiquez l'emplacement du fichier et observez les temps passés dans chaque fonction du programme. Quelle est l'opération dominante, c'est-à-dire celle qui prend le plus de temps d'exécution ?

## 6. Compter le nombre d'appels de l'opération dominante

Dans cet algorithme, la fonction wordncmp est utilisée dans trois buts :
- initialisation du tableau word
- recherche dichotomique des préfixes candidats dans le tableau
- choix aléatoire d'un préfixe parmi les préfixes candidats

Nous allons compter le nombre d'appels à cette fonction dans les trois phases. Il faut pour cela éditer légèrement le code. Définissez trois variables count1, count2, count3 et incrémentez-les quand il le faut. Affichez leurs valeurs à la fin du programme. Affichez également la somme des 3.

Rappel : si x est de type unsigned long, alors on l'affichera dans la sortie d'erreur de la façon suivante : fprintf(stderr, « x vaut %lu\n », x) ;

Voir la page http://en.wikipedia.org/wiki/Standard_streams si vous ne connaissez pas la notion de sortie d'erreur.

## 7. Mesurer la complexité de l'algorithme en fonction de n, m et k

On veut tester l'influence de *n, m* et *k* sur la valeur de count3. Concevez un plan d'expérience, écrivez-le sur un « cahier de laboratoire » (papier ou numérique) en vous inspirant de l'exemple montré en cours. Faîtes les tests et sauvegardez les résultats dans un tableur.

Indication : si vous souhaitez utiliser d'autres fichiers de départ, vous pouvez télécharger des œuvres au format txt sur le site du projet Gutenberg. Enlevez l'entête Gutenberg au début du fichier et la licence Gutenberg à la fin du fichier.