Compte-rendu du TP
“Évaluation empirique d’un programme”

Adrien Castex, Polytech Informatique et Master Recherche IA
Benoit Vuillemin, Polytech Informatique et Master Recherche IA

25 janvier 2016

Table des matières

1 Présentation sommaire du programme à évaluer

1.1 Données à fournir en entrée

On fournit, en entrée, un fichier texte venant d'un roman. Ce fichier contient uniquement le texte du roman. Il ne doit pas avoir de numéro de page ou d'éléments qui ne sont pas liés au texte lui-même.

1.2 Données fournies en sortie

En sortie, nous obtenons un texte composé de m mots (si possible) issus du texte d'entrée. Le résultat est une texte qui donne l'illusion d'avoir été rédigé par une personne. Cette illusion va être plus ou moins efficace en fonction du paramètre k. Une valeur trop grande du paramètre k va donner un texte identique au texte d'origine.

1.3 Algorithme

L'algorithme découpe le texte d'entrée en mots et les trie. L'algorithme va successivement trouver l'index du mot anciennement utilisé, jusqu'à ce qu'il n'y ait plus de mots disponibles ou jusqu'à ce qu'il ait atteint le nombre de mots de sortie désiré. Ensuite, il sélectionne un nouveau mot à partir de cet index, et pour terminer, il écrit ce mot dans la sortie désirée.

— n représente le nombre de mots mesurés dans le fichier d'entrée.
— m représente le nombre de mots que l'on souhaite avoir en sortie.
— k représente le nombre de mots par phrase.

L'algorithme est stochastique dans sa sélection des mots. Néanmoins, sa stochasticité peut être annulée par l'utilisation d'une mauvaise graine pour l'initialisation de la fonction d'aléatoire (srand).

2 Premières mesures du temps d'exécution

2.1 Paramètres utilisés dans toute cette section

2.1.1 Paramètres

Étant donné que les tests n'ont pas été réalisés avec les mêmes paramètres à chaque fois, ceux-ci seront indiqués.

2.1.2 Environnement de travail

Table 1 – Informations sur la machine utilisée

OS	Microsoft Windows 10 Professionnel 10.0.10240
Processeur	Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz 4 coeurs 8 processeurs logiques
Memoire	7.66GB DDR3
Disque	1TB HDD Toshiba MQ01ABD100
Compilateur	gcc 4.8.1 (Windows 10)

2.2 Mise en évidence de la stochasticité de l'algorithme

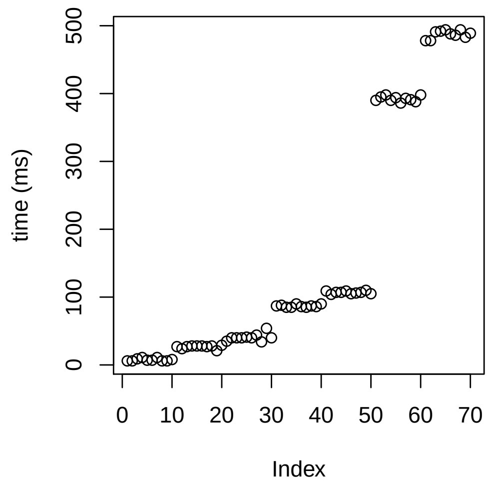

FIGURE 1 – Mesure du temps CPU - m = 10000, k = 2

Nous pouvons voir sur la figure 1 différents paliers représentant les différents fichiers d'entrée. Chaque temps mesuré d'un palier (donc d'un fichier) est une exécution identique les unes par rapport aux autres. Nous voyons donc qu'au sein d'un même palier, les temps varient, représentant la stochasticité de l'algorithme.

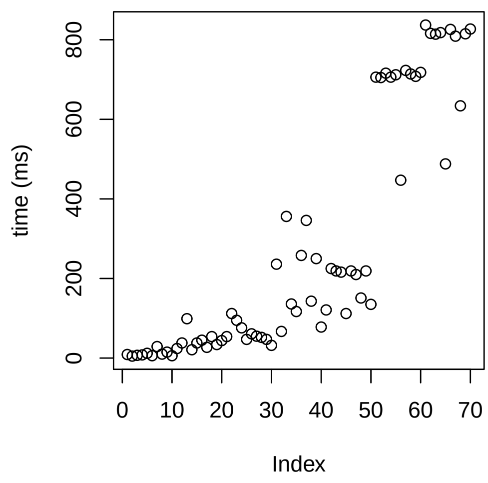

FIGURE 2 – Mesure du temps CPU - m = 100000, k = 2

Sur la figure 2, nous pouvons voir une stochasticité plus importante, sûrement dû à une valeur de m 10 fois plus grande.

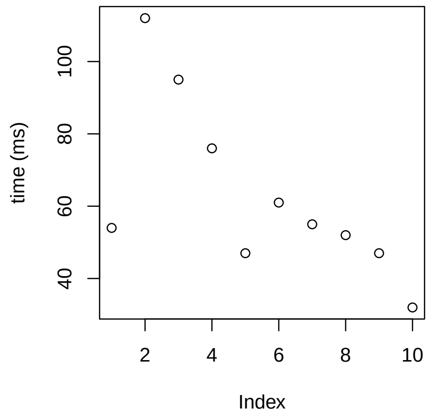

FIGURE 3 – Mesure du temps CPU - m = 100000, k = 2, n = 39213

La figure 3 montre la stochasticité de l'algorithme sur un seul palier. Ceci veut dire que tous les points représentent le temps d'exécution avec des paramètres identiques.

TABLE 2 – Valeurs de n choisies

m	k	n	Ecart-type
100000	2	39213	24.35136 ms
100000	7	39213	31.43317 ms
100000	*	39213	29.36821 ms

Ces tests ont été réalisés en 10 runs.

Nous avons appliqué à *srand* une valeur fixe (ici, 0). Ainsi, à chaque exécution de mêmes paramètres, les résultats seront très proches les uns des autres. Cela peut être clairement vu sur la figure 4, avec, en noir, 100 exécutions avec une graine qui change, et en rouge, 100 exécutions avec une graine constante :

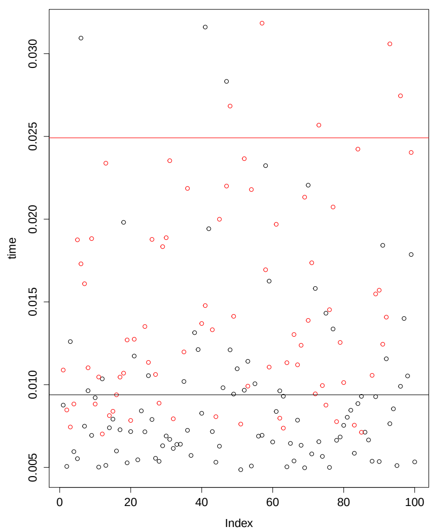

FIGURE 4 – Temps CPU - srand fixe

2.3 Influence de la charge système

Le temps CPU est influencé par la charge système car si celui-ci est très demandé par d'autres processus, alors il y a plus de chance pour que le processeur soit coupé entre deux mesures de temps pour donner la main à un autre processus.

Pour observer ce phénomène, il suffit de lancer les tests en série et de les comparer aux mêmes tests lancés en parallèle. Étant donné que le système va devoir passer d'un processus à l'autre, nous observerons donc des différences.

100 tests ont été faits, avec comme paramètres k=3, m=100000 et, comme fichier, Le catéchumène (6640 mots).

Pour pouvoir observer en grande précision, les tests ont été faits en mesurant non pas le temps, mais un compteur incrémenté par le système (qui représente le temps lorsqu'il est divisé par la fréquence du processeur), ce qui a été fait pour être facilement analysable par des êtres humains.

Table 3 – Résultats

	Série	Parallèle
Écart-type	0.005357606	0.02199298
Minimum	0.004862	0.007019
Maximum	0.031609	0.097435
Moyenne	0.00938539	0.0249153
Médiane	0.0075155	0.0155945

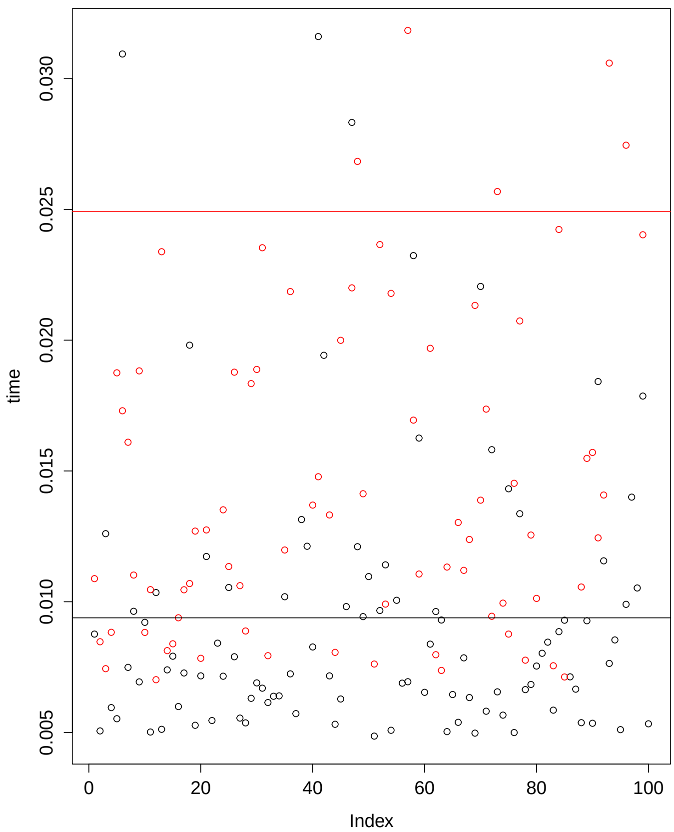

Figure 5 – Temps d'exécution en parallèle

En rouge, nous pouvons voir le temps d'exécution en parallèle. En noir, nous pouvons voir le temps d'exécution en série. Les lignes représentent les moyennes.

2.4 Influence des options de compilation

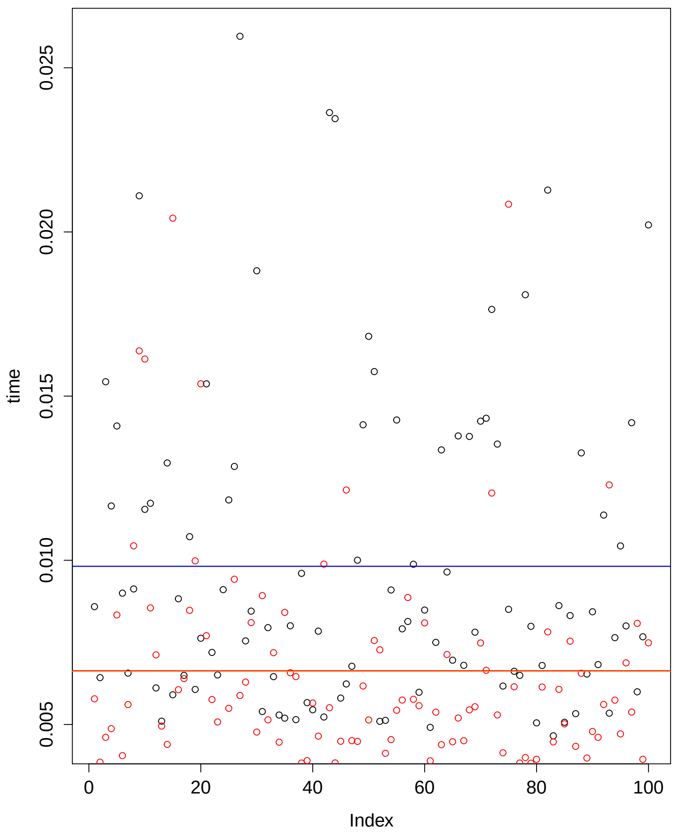

FIGURE 6 – Temps d'exécution en fonction de l'optimisation

Nous pouvons constater sur la figure 6 que si l'optimisation n'a pas été activée (noir) ou si elle a été activée au niveau 1 (bleu), cela n'implique aucun changement de temps d'exécution. Mais à partir de l'optimisation de niveau 2 (orange), on constate une amélioration. L'optimisation de niveau 3 (rouge) n'implique aucune amélioration par rapport au niveau 2.

$moyenne(tempsDebug) = 0.00981731$

$moyenne(tempsRelease) = 0.006638$

$moyenne(tempsRelease)/moyenne(tempsDebug) = 0.6761526$

=> Amélioration de 67.61526 %

Il y a eu 100 runs pour chaque paramètre d'optimisation, avec pour paramètres k=3, m=100000 et, comme fichier, Le catéchumène (6640 mots). Les optimisations sont : 0, 1, 2, 3

3 Opération dominante

3.1 Paramètres utilisés dans toute cette section

Pour cette section, les paramètres utilisés sont les suivants : k = 3, m = 10000 et, comme fichier, Don Quixiote (404461 mots).

3.2 Identification de l'opération dominante

Pour déterminer l'opération dominante au sein du programme, nous avons utilisé gprof.

```
$ gcc -pg -o markovPG markov.c
$ ./markovPG.exe 3 10000 < ./don-quixote.txt > outPG.txt
$ gprof markovPG.exe gmon.out > pg2.out
```

```
%    cumulative   self              self     total
time   seconds   seconds    calls  ns/call  ns/call  name
53.57      0.15     0.15  7746432    19.36    19.36  wordncmp
17.86      0.20     0.05                             sortcmp
14.29      0.24     0.04                             __fentry__
14.29      0.28     0.04                             _mcount_private
0.00      0.28     0.00    30000     0.00     0.00  skip
0.00      0.28     0.00    10000     0.00     0.00  writeword
```

Nous pouvons voir que la fonction *wordncmp* est la plus appelée, avec un total de 7746432 appels.

3.3 Comptage des appels à l'opération dominante

Nous avons ajouté, de façon globale, un tableau de 3 cases contenant des long int. Nous incrémentons ensuite, à chaque appel, la case du tableau correspondante.

```
/* [...] */
up = nword;
while(lo+1 != up)
{
  mid = (lo + up) / 2;
  nbCall[0]++;
  if(wordncmp(word[mid], phrase) < 0)
    lo = mid;
  else
    up = mid;
}
/* [...] */
```

```
/* [...] */
nbCall[1]++;
for(i = 0; wordncmp(phrase, word[up+i]) == 0; i++)
{
  if(rand() % (i+1) == 0)
    p = word[up+i];
  nbCall[1]++;
}
/* [...] */
```

Il ne faut pas oublier de mettre une incrémentation avant ou après la boucle for car celle-ci va faire x tours de boucle, impliquant d'avoir x fois la condition qui vaut *vrai* et 1 fois qui vaut $faux$. Nous avons donc $x + 1$ appels à *wordncmp*.

```
/* [...] */
/* called by system qsort */
int sortcmp(const void* p, const void* q)
{
  char** p1 = (char**)p;
  char** q1 = (char**)q;
  nbCall[2]++;
  return wordncmp(*p1, *q1);
}
/* [...] */
```

4 Analyse de l'expérience fournie par R. Thion

Nous souhaitons tester l'influence de n, m et k sur les valeurs de $count1$, $count2$ et $count3$.

4.1 Plan d'expérience

4.1.1 Variables mesurées

Les variables mesurées sont les suivantes :

Table 4 – Variables mesurées

Variable	Description
count1	Nombre d'appels à la méthode *wordncmp*.
count2	Nombre d'appels à la méthode *wordncmp*.
count3	Nombre d'appels à la méthode *wordncmp*.
usr	Temps approximatif utilisé par le programme en lui même.
sys	Temps approximatif utilisé par le système d'exploitation.
time	Temps approximatif entre le début et la fin de l'exécution du programme.

4.1.2 Paramètres

Il a été réalisé 20 tests par jeu de paramètre.

Table 5 – Paramètres numériques

Paramètre	Valeur min	Valeur max	Incrément
m	100	1000000	$*10$
k	2	7	$+1$

Table 6 – Valeurs de n choisies.

Fichier	Nombre de mots
book1_assommoir.txt	91033
book2_sherlock.txt	104410
book3_wonderland.txt	26438
book4_don-quixote_400k.txt	404461
book4_don-quixote_40k.txt	40171
book4_don-quixote_4k.txt	4030
book5_allbooks.txt	626342

4.1.3 Mode de combinaison des valeurs

Les tests ont été effectués à partir de plages de paramètres. Toutes les valeurs possibles n'ont pas été utilisées, étant donné que certaines valeurs ne sont plus adaptées à une utilisation cohérente du programme, par exemple : de grandes valeurs de k. Cela veut dire que l'utilisation de tels paramètres donnerait un résultat d'exploitation (et non en terme de temps, etc...) non désiré dans une utilisation normale du programme.

4.1.4 Nombre de runs

4200

4.1.5 Environnement de test

Table 7 – Informations sur la machine utilisée

OS	4.2.0-23-generic #28-Ubuntu SMP x86_64 GNU/Linux (Ubuntu 15.10)
Processeur	Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz (dual-core)
Memoire	8GB 1600MHz DDR3L
Disque	180GB M2 SATA-3 Solid State Drive
Compilateur	g++ (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010

4.2 Analyse des résultats

Count1 ∼ n + k + m

```
Estimate Std. Error  t value Pr(>|t|)
(Intercept) -1.756e+05  4.624e+03  -37.980   <2e-16 ***
n            1.811e+01  7.032e-03 2575.692   <2e-16 ***
k            3.159e+00  9.036e+02    0.003    0.997
m            9.230e-16  3.950e-03    0.000    1.000
```

Nous pouvons voir que seul n est fortement lié à Count1.
R^2 de $count1 \sim n$: 0.9994

Count2 ∼ n + k + m

```
Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.487e+05  8.466e+04  -4.119 3.88e-05 ***
n            2.988e+00  1.287e-01  23.207  < 2e-16 ***
k           -2.293e+03  1.654e+04  -0.139     0.89
m            2.206e+00  7.231e-02  30.505  < 2e-16 ***
```

Nous pouvons voir que seul n et m sont fortement liés à Count2.
R^2 de $count2 \sim n$: 0.09485
R^2 de $count2 \sim m$: 0.1641

Count3 ∼ n + k + m

```
Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.134e+06  2.007e+05   5.652 1.69e-08 ***
n            3.291e+00  3.051e-01  10.785  < 2e-16 ***
k           -3.658e+05  3.921e+04  -9.328  < 2e-16 ***
m            1.848e+00  1.714e-01  10.780  < 2e-16 ***
```

Nous pouvons voir que n, k et m sont étroitement liés à Count3.
R^2 de $count3 \sim n$: 0.02553
R^2 de $count3 \sim k$: 0.01904
R^2 de $count3 \sim m$: 0.0255

Coefficients de corrélation :

```
count1       count2     count3
n 9.996839e-01  0.308325932  0.1605025
k 1.356917e-06 -0.001841625 -0.1388157
m 1.115266e-19  0.405293749  0.1604183
```

En regardant les résultats obtenus sur les jeux de paramètres identiques, nous pouvons constater que les résultats obtenus sont les mêmes. Cela peut s'expliquer si les tests ont été réalisés avec une graine aléatoire ($srand(...)$) en seconde telle que $time(NULL)$. En effet, cette dernière fonction retourne le temps en seconde, ce qui fournit à $srand$ la même valeur tant que les tests sont réalisés la même seconde. Étant donné que l'exécution du programme a nécessité peu de temps et que ceux-ci ont été enchainés en moins d'une seconde, ceci explique ce résultat. Par conséquent, il est préférable d'utiliser une graine qui dépend du nombre de cycles du processeur, et non du temps, lorsque l'on effectue de nombreux tests à la suite. Nous pouvons donc conclure qu'environ 210 résultats sur 4200 sont réellement intéressants, les autres étant des doublons.

5 Conception et réalisation de notre expérience

Nous souhaitons tester l'influence de n, m et k sur les valeurs de $count1$, $count2$ et $count3$.

5.1 Plan d'expérience

Pour réaliser les expériences et éviter le problème présenté dans l'analyse des résultats de l'expérience fournie vis à vis de l'initialisation de l'algorithme de génération de nombres aléatoires, nous avons utilisé une fonction de la librairie *windows.h* qui retournera une valeur différente à chaque appel.

```
#include <windows.h>

/* [...] */

unsigned __int64 rndSeed;
QueryPerformanceCounter((LARGE_INTEGER *)&rndSeed);
srand(rndSeed);

/* [...] */
```

5.1.1 Variables mesurées

Les variables mesurées sont les suivantes :

Table 8 – Variables mesurées

Variable	Description
count1	Nombre d'appels à la méthode *wordncmp*
count2	Nombre d'appels à la méthode *wordncmp*
count3	Nombre d'appels à la méthode *wordncmp*
usr	Temps approximatif utilisé par le programme en lui même
sys	Temps approximatif utilisé par le système d'exploitation
time	Temps approximatif entre le début et la fin de l'exécution du programme

5.1.2 Paramètres

Il a été réalisé 10 tests par jeu de paramètre.

Table 9 – Paramètres numériques

Paramètre	Valeur min	Valeur max	Incrément
m	100	1000000	$*10$
k	2	7	$+1$

Table 10 – Valeurs de n choisies.

Fichier	Nombre de mots
Le catéchumène	6640
Zig ou la destinée	26046
Venus Boy	39213
Battlefields of the Marne	75166
Madame Bovary	112451
Don Quixote	404461
Histoire des salons de Paris	492732

5.1.3 Mode de combinaison des valeurs

Les tests ont été effectués à partir de plages de paramètres. Toutes les valeurs possibles n'ont pas été utilisées, étant donné que certaines valeurs (par exemple : de grandes valeurs de k) ne sont plus adaptées à une utilisation cohérente du programme. C'est à dire que l'utilisation de tels paramètres donnerait un résultat d'exploitation (et non en terme de temps, etc...) non désiré dans une utilisation normale du programme.

5.1.4 Nombre de runs

Nous avons réalisé 2100 runs.

5.1.5 Environnement de test

Table 11 – Informations sur la machine utilisée

OS	Microsoft Windows 10 Professionnel 10.0.10240
Processeur	Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz 4 coeurs 8 processeurs logiques
Memoire	7.66GB DDR3
Disque	1TB HDD Toshiba MQ01ABD100
Compilateur	gcc 4.8.1 (Windows 10)

5.2 Analyse des résultats

Count1 ∼ n + k + m

```
Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.856e+05  2.629e+05  -2.608  0.00917 **
n            1.209e+01  4.757e-01  25.421  < 2e-16 ***
k            7.583e+04  5.109e+04   1.484  0.13789
m            1.718e-02  2.233e-03   7.694 2.18e-14 ***
```

Nous pouvons voir que seul n est fortement lié à Count1.
R^2 de $count1 \sim n$: 0.2301

Count2 ∼ n + k + m

```
Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.753e+05  7.855e+03  -35.05   <2e-16 ***
n            2.247e+01  1.422e-02 1580.88   <2e-16 ***
k            1.853e+04  1.527e+03   12.14   <2e-16 ***
m            4.283e-17  6.673e-05    0.00        1
```

Nous pouvons voir que seul n et k sont fortement liés à Count2.
R^2 de $count2 \sim n$: 0.9991
R^2 de $count2 \sim k$: -0.0004177

Count3 $\sim$ n + k + m

```
Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.356e+06  3.457e+05   9.708  < 2e-16 ***
n            8.389e+00  6.256e-01  13.409  < 2e-16 ***
k           -8.403e+05  6.719e+04 -12.506  < 2e-16 ***
m            1.241e-02  2.937e-03   4.225 2.49e-05 ***
```

Nous pouvons voir que n et k sont étroitement liés à Count3.
R^2 de $count3 \sim n$: 0.07295
R^2 de $count3 \sim k$: 0.06339

Coefficients de corrélation :

```
count1        count2        count3
n 0.48010732  9.995515e-01  0.27090054
k 0.02803182  7.675098e-03 -0.25266565
m 0.14530334 -2.633931e-20  0.08536216
```

Nous n'obtenons pas les mêmes résultats que l'expérience fournie. Cela peut venir de la stochasticité du programme, du problème d'initialisation de l'algorithme aléatoire pour l'expérience fournie et/ou des fichiers utilisés qui peuvent avoir des caractéristiques particulières.

6 Annexe : Code réalisé pour le tutoriel sur les tests d'hypothèse

Algorithm 1 Algorithme du générateur de texte.

```
nword ← splitInputIntoWords(word)                ▷ Découper le fichier d'entrée en mots
sort(word)
phrase ← inputchars
wordsleft ← m
for wordsleft > 0 do              ▷ Jusqu'à ce que l'on ait tous les mots que nous voulions.
              ▷ Dichotomie pour trouver l'index du mot qui correspond le mieux à phrase.
    lo ← −1                                      ▷ Début du tableau de mots d'entrée.
    up ← nword                                   ▷ Fin du tableau de mots d'entrée.
    while lo + 1 ≠ up do
        mid ← (lo + up)/2                        ▷ mid = médiane entre lo et up.
        if isSmallerThan(word[mid], phrase) then
            lo ← mid
        else
            up ← mid
        end if
    end while
    i ← 0
    for phrase = word[up + i] do
        if !(rand() % (i + 1)) then ▷ Se produit de moins en moins souvent au fur et à mesure que
la valeur de i augmente.
            p ← word[up + i]
        end if
        i ← i + 1
    end for
    phrase ← skip(p, 1)                          ▷ Sélectionne le mot choisi aléatoirement.
    if isEmptyString(skip(phrase, k-1)) then     ▷ Si nous avons atteint la fin du fichier.
        break                                    ▷ Quitte la boucle.
    end if
    writeOutputWord(skip(phrase, k − 1))
    wordleft ← wordleft − 1     ▷ Décompte le nombre restant de mots à ajouter au texte final.
end for
```

```
#######################
# Cas ou H0 est vraie
#######################

# On va considerer deux populations qui ont exactement
# les memes parametres : les deux sont uniformement
# distribuees entre 0 et 1. Les deux populations ont
# donc la meme moyenne, en l'occurrence 0.5.
# Puis nous allons faire une experience :
# - tirer un petit echantillon dans chaque population
# - calculer la moyenne observee dans l'echantillon 1,
#   puis celle observee dans l'echantillon 2. On
#   n'obtiendra pas exactement 0.5 ni pour l'une ni pour
#   l'autre.
# - calculer la difference entre les deux moyennes
#   d'echantillons. Appelons d cette difference.

tirerEchantillonPop1H0 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

tirerEchantillonPop2H0 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

# Ecrire ici le code permettant de faire une "experience"
# c'est a dire tirer un echantillon de taille 20 dans
# chaque population et mesurer la difference d observee
# entre les 2 moyennes d'echantillon.
pop1 <- tirerEchantillonPop1H0(20)
pop2 <- tirerEchantillonPop2H0(20)

mean(pop1)
mean(pop2)
abs(mean(pop1) - mean(pop2))
abs(mean(tirerEchantillonPop1H0(20)) - mean(tirerEchantillonPop2H0(20)))

# Si 1000 etudiants font independamment cette experience,
# chaque etudiant aura des echantillons differents et donc
# chacun aura une valeur differente pour d. Si nous
# mettons toutes ces 1000 valeurs ensemble dans un vecteur
# (appele diffMoyH0), quelle sera la moyenne de ce vecteur ?

diffMoyH0 = c()

for(i in  1:1000)
{
  diffMoyH0 = c(diffMoyH0, abs(mean(tirerEchantillonPop1H0(20)) - mean(tirerEchantillonPop2H0
      (20))))
}

# On affiche la moyenne totale.
mean(diffMoyH0)

# On affiche les 1000 valeurs sur un histogramme.
plot(diffMoyH0)
abline(h=mean(diffMoyH0), col="red")
```

Nous obtenons une moyenne de 0,07.

```r

######################
# Cas ou H1 est vraie
######################

# On va voir comment se comporte notre indicateur quand
# les deux populations n'ont pas la meme moyenne
# (mais ont tout de meme la meme dispersion)
# eg . pop1 comprise entre 0.0 et 1.0
#   et pop2 comprise entre 0.2 et 1.2

tirerEchantillonPop1H1 <- function(n) {
  vraieMoyenne <- 0.5
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}

tirerEchantillonPop2H1 <- function(n) {
  vraieMoyenne <- 0.7
  plageDeDispersion <- 1.0
  min <- vraieMoyenne - plageDeDispersion/2.0
  max <- vraieMoyenne + plageDeDispersion/2.0
  ech <- runif(n, min, max)
  return(ech)
}


# Ecrire ici le code pour faire les 1000 experiences
# comme precedemment, en nommant cette fois le vecteur
# diffMoyH1. Calculez sa moyenne et tracez son histogramme.

diffMoyH1 = c()

for(i in  1:1000)
{
  diffMoyH1 = c(diffMoyH1, abs(mean(tirerEchantillonPop1H1(20)) - mean(tirerEchantillonPop2H1
      (20))))
}

# On affiche la moyenne totale.
mean(diffMoyH1)

# On affiche les 1000 valeurs sur un histogramme.
plot(diffMoyH1)
abline(h=mean(diffMoyH1), col="red")
```

Nous obtenons une moyenne de 0,2.

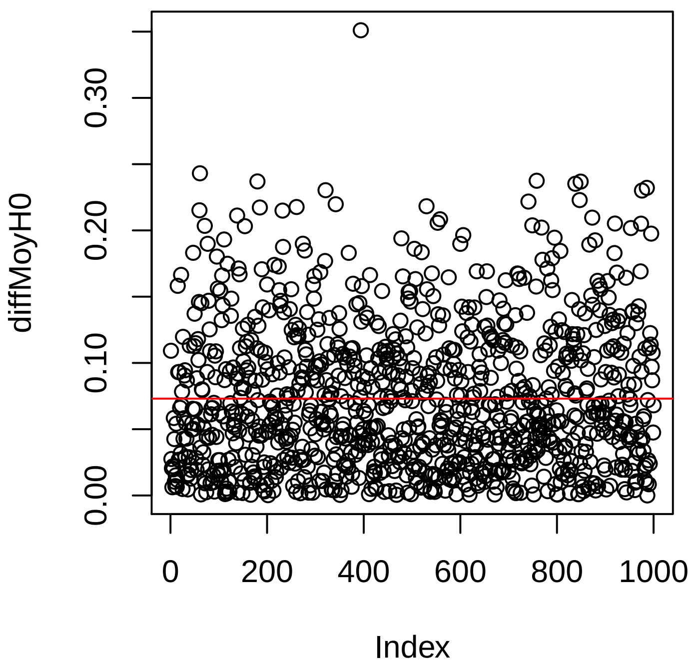

FIGURE 7 – Graphique représentant diffMoyH0 et diffMoyH1 ainsi que leur moyenne

Plaçons nous maintenant dans le cas où l'on ait fait une seule expériencee.

```
cMean <- function(v1, v2)
{
  return(abs(mean(v1) - mean(v2)))
}
sameMean <- function(v1, v2)
{
  return(cMean(v1, v2) < 0.1)
}

sameMean(tirerEchantillonPop1H0(20), tirerEchantillonPop2H0(20))
sameMean(tirerEchantillonPop1H1(20), tirerEchantillonPop2H1(20))

x1 <- c(0.97050525, 0.13734516, 0.80793033, 0.05207726, 0.62629180, 0.93485856,
0.58220744, 0.65935145, 0.76467195, 0.73512414, 0.45139560, 0.93225380,
0.53790595, 0.99845675, 0.31035081, 0.43082815, 0.15475353, 0.42647652,
0.65676067, 0.74186048)
x2 <- c(0.33565036, 0.28830545, 0.51556544, 0.93223089, 0.29192576, 0.43505823,
0.63127002, 0.86082799, 0.56533392, 0.19083212, 0.13087779, 0.09849703,
0.98921291, 0.91480756, 0.78556552, 0.33859160, 0.88482223, 0.76701274,
0.24190609, 0.46251866)

cMean(x1, x2)
sameMean(x1, x2)
```

Ici, sameMean(x1, x2) retourne "TRUE", ce qui signifie que les deux échantillons ont une moyenne proche l'une de l'autre.

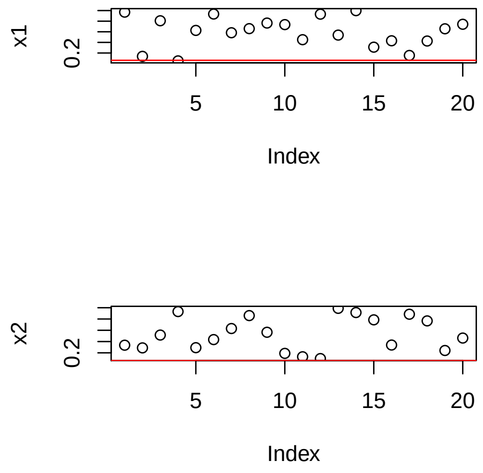

FIGURE 8 – Graphique représentant x1 and x2 ainsi que leur moyenne

```
x1 <- c(0.41236444, 0.28422821, 0.15093798, 0.05885328, 0.25514435, 0.63026931,
0.56325462, 0.76304859, 0.56523993, 0.92535660, 0.10898729, 0.51579642,
0.07223967, 0.53483839, 0.52516575, 0.20250815, 0.89634680, 0.53879059,
0.58736912, 0.53945749)
x2 <- c(1.0809923, 0.5772333, 0.7252340, 1.1529082, 1.0924642, 0.6046166, 0.9495800,
0.3019857, 0.7701195, 1.0746508, 0.3928894, 1.0885017, 0.5101510, 1.1871599,
0.7953318, 0.5711237, 0.5505642, 0.9854085, 0.5105643, 0.9601635)

cMean(x1, x2)
sameMean(x1, x2)
```

Ici, sameMean(x1, x2) retourne "FALSE", ce qui signifie que les deux échantillons ont une moyenne bien différente l'une de l'autre.

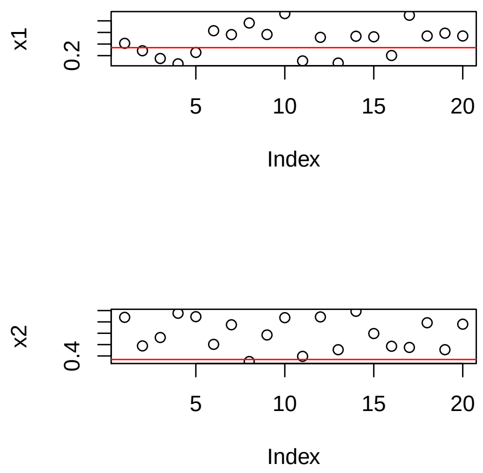

FIGURE 9 – Graphique représentant x1 and x2 ainsi que leur moyenne