TP Algorithme expérimentale

TURPIN Pierre

28 novembre 2014

Table des matières

1 Mesure du temps d'exécution du programme

1.1 Découverte de l'aspect stochastique du programme

La compilation de la source est faite en mode débug avec l'option -g de gcc.

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

Les options d'exécutions du programme seront k = 3 et m = 1 000 000 en utilisant le fichier texte *don-quixote.txt*.

L'éxécution avec ces paramètres est lancées 3 fois.
Les temps d'exécution de celle-ci sont mesurés par la command *time* et sont enregistrés dans le même fichier.
Les résultats d'exécution sont également stockés dans des fichiers différents.

```
rm -f ./t

/usr/bin/time -a -o ./t -f "%e" ./markov 3 1000000 < ./don-quixote.txt > ./tmp1
/usr/bin/time -a -o ./t -f "%e" ./markov 3 1000000 < ./don-quixote.txt > ./tmp2
/usr/bin/time -a -o ./t -f "%e" ./markov 3 1000000 < ./don-quixote.txt > ./tmp3
```

Dans le fichier de temps d'exécution, on remarque que les trois lignes diffèrent.
Cela signifie que la durée de chaque exécution est différentes.

Les fichiers de résultats contiennent chacun des textes complètement différent.

Ces différences doivent être du à une instruction stochastique dans le programme.

1.2 Mesure du temps d'exécution avec k=3, m=1000000

Le programme est lancé 100 fois avec les même paramètres avec les fichiers d'entrées *don-quixote.txt*, *madame-bovary.txt* et *zadig.txt*. Les résultats sont ignorés dans cette étude. Seul les temps d'exécutions sont enregistrés dans un fichier par texte.

```
# Nettoyage des fichiers de temps d'execution
rm -f ./tD
rm -f ./tM
rm -f ./tZ

for i in $(seq 1 100); do
  /usr/bin/time -a -o ./tD -f "%e" ./markov 3 1000000 < ./don-quixote.txt > /dev/null
done

for i in $(seq 1 100); do
```

```
  /usr/bin/time -a -o ./tM -f "%e" ./markov 3 1000000 < ./madame-bovary.txt > /dev/null
done

for i in $(seq 1 100); do
  /usr/bin/time -a -o ./tZ -f "%e" ./markov 3 1000000 < ./zadig.txt > /dev/null
done
```

Les temps d'exécution sont visible sur l'histogramme 1. Le texte de Don Quixote prend beaucoup plus de temps que les autres.
On voit pour chaque texte que le temps de l'exécution est différentes à chaque lancement. Cela montre l'aspect stochastique du programme.

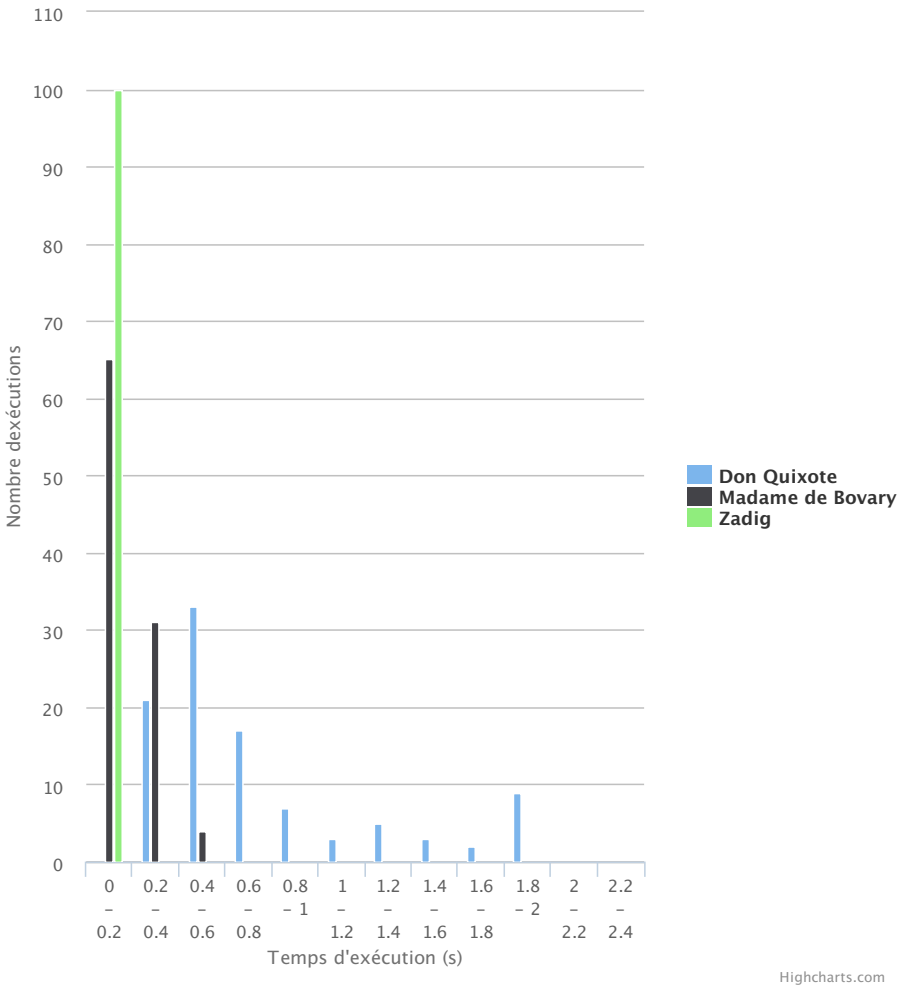

FIGURE 1 – Temps d'exécution du programme avec k=3 et m=1000000

1.3 Suppression de l'aspect stochastique

Le programme utilise la fonction *rand* de la bibliothèque C. Cette fonction est une interface au générateur de nombre pseudo aléatoire de la bibliothèque. Ce générateur s'initialise par une *seed* avec la fonction d'API *srand*.

Pour supprimer l'aspect aléatoire du programme, l'appel à *srand* dans le code est commenté.

Plusieurs exécutions avec les même paramètres rendent exactement le même résultat et ont un temps d'exécution très semblable (< à 2ms de différences).

1.4 Influence de la charge CPU

Pour vérifier si la charge du CPU influe sur le temps d'exécution, 20 exécutions pour chaque texte sont lancées en parallèles. On s'attend alors à voir que ceux utilisant le texte de Zadig sont ralenties par les autres exécutions.

```
# Nettoyage des fichiers de temps d'execution
rm -f ./tDC
rm -f ./tMC
rm -f ./tZC

for i in $(seq 1 20); do
  /usr/bin/time -a -o ./tDC -f "%e" ./markov 3 1000000 < ./don-quixote.txt > /dev/null &
done
for i in $(seq 1 20); do
  /usr/bin/time -a -o ./tMC -f "%e" ./markov 3 1000000 < ./madame-bovary.txt > /dev/null &
done
for i in $(seq 1 20); do
  /usr/bin/time -a -o ./tZC -f "%e" ./markov 3 1000000 < ./zadig.txt > /dev/null &
done
```

Les temps d'exécution sont visible sur l'histogramme 2. Les temps d'exécution sont clairement ralenties par rapport à la première étude. Cela est du à la prise de CPU par chacune des instances du programme.

1.5 Optimisation par le compilateur

Le programme est recompilé en mode release optimisé en utilisant l'option *-O3* de *gcc* :

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

L'étude du temps d'exécution est à nouveau effectué seulement sur le fichier de Don Quixote. Les 100 exécutions ne seront pas lancées en parallèles.

```
# Nettoyage des fichiers de temps d'execution
rm -f ./t03

for i in $(seq 1 100); do
  /usr/bin/time -a -o ./t03 -f "%e" ./markov 3 1000000 < ./don-quixote.txt > /dev/null
done
```

Les temps sont comparés à ceux des exécutions sans le paramètre d'optimisation. Le graphique 3 montre bien que l'optimisation accélère l'exécution du programme.

On obtient un temps médian de $0.34s$ avec l'optimisation et $0.54s$ sans. Le pourcentage d'amélioration est donc de l'ordre de 37%

2 Trouver l'opération dominante

Pour trouver l'opération dominante, le programme est recompilé en mode débug avec l'option *-pg*.

L'exécutable est lancé une fois avec des paramètres fixes puis l'exécution peut être analysé grâce au fichier *gmon.out* avec le programme *gprof*.

```
gcc -pg -o markov ./markov.c
./markov 3 1000000 < ./don-quixote.txt > /dev/null
gprof ./markov ./gmon.out
```

Le résultat de *gprof* est visible ci-dessous :

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 65.97      0.32     0.32 10653342    29.72    29.72  wordncmp
 18.85      0.41     0.09                             main
  4.19      0.43     0.02   472367    42.56    42.56  skip
  3.14      0.44     0.02                             sortcmp
  0.00      0.44     0.00   157455     0.00     0.00  writeword
```

La fonction la plus appelée dans le programme et qui prend le plus de temps d'exécution est incontestablement *wordncmp*.
Elle prend 65.97% du temps d'exécution du programme.

L'exécution a été faite plusieurs fois pour pouvoir vérifier qu'on obtient le même résultat malgré l'aspect stochastique du programme.

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

La fonction *wordncmp* est utilisé dans trois buts :

— initialisation du tableau word
— recherche dichotomique des préfixes candidats dans le tableau
— choix aléatoire d'un préfixe parmi les préfixes candidats

Chacune des utilisations est profilé en plaçant dans le code trois compteurs. Ces trois compteurs sont ensuites affichés séparés par des espaces sur une ligne de *stderr*.

On remarque que l'exécution rend des résultats différents. Le programme est lancé 100 fois avec comme paramètres ($k = 3$, $m = 1000000$, text=./don-quixote.txt) pour pouvoir récupérer les valeurs médiannes.

```
# Nettoyage des fichiers de comptage
rm -f ./count

for i in $(seq 1 100); do
  ./markov 3 1000000 < ./don-quixote.txt > /dev/null 2>> ./count
done
```

Les résultats donnent :

7024519 appels médians pour l'initialisation du tableau word

5243404 appels médians pour la recherche dichotomique des préfixes candidats dans le tableau

936562 appels médians pour le choix aléatoire d'un préfixe parmi les préfixes candidats

On remarque que l'initialisation du tableau word n'est pas atteint par l'aspect stochastique. Le nombre d'appel à *wordncmp* dans cette partie est donc constant.

Au total, il y a 13204485 appels médians pour la fonction *wordncmp*.

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

4.1 Protocole expérimental

Pour plus de rapidité de test, le programme est compilé avec l'option d'optimisation *-O3* du compilateur.

Dans cette étude, la variable *count3* ainsi que le temps d'exécution sont enregistrés. Pour voir l'évolution des résultats en fonction des variations des trois variables k, m, n, plusieurs sous-études ont été menées.

Une sous-étude correspond à une configuration particulière.
Les trois textes sont utilisés donc $n \in \{404461, 112972, 26046\}$ pour, respectivement, Don Quixote, Madame de Bovary et Zadig.
La variable k est défini entre 2 et 20 inclus [1] [2].
La variable m est testé entre 100000 et 1000000 avec un pas de 100000.
Enfin comme chaque exécution est aléatoire, une sous-étude correspond à lancer 100 fois le programme avec la même configuration d'entrée.
Les valeurs résultantes de temps d'exécution et de *count3* sont alors les moyennes des 100 exécutions.

Ce protocole d'exécution est effectué par ce script suivant :

```
#!/usr/bin/env sh

dir="./results"
rm -rf $dir
mkdir $dir

gcc -O3 -o ./markov ./markov.c

# Configuration de k
kMin=2
kMax=20
kStep=1

# Configuration de m
mMin=100000
mMax=1000000
mStep=100000

# Nombre d'exec par configuration
N=100

# Liste des textes
```

1. Les temps d'exécutions étaient trop grand pour k=1
2. Sur certains graphiques, k=2 n'est pas montré car le temps d'exécution ou *count3* étaient trop grand ce qui rendait illisible les variations sur le reste du graphique.

```
texts="don-quixote.txt madame-bovary.txt zadig.txt"

for txt in $texts; do
  n=$(cat $txt | wc -w)
  for k in $(seq $kMin $kStep $kMax); do
    for m in $(seq $mMin $mStep $mMax); do
      echo "$N executions avec k=$k, m=$m, n=$n, txt=$txt"
      for i in $(seq 1 $N); do
        /usr/bin/time -a -o "$dir/time-$txt-$k-$m-$n" -f "%e" \
          ./markov $k $m < $txt > /dev/null 2>> "$dir/count-$txt-$k-$m-$n"
      done
    done
  done
done
```

Les résultats sont des simples fichiers contenants 100 lignes chacun. Chaque fichier est ensuite lu pour être moyenné puis structuré pour ensuite être transformé en configuration de dataviz Highcharts.com (heatmap ou courbes) [3].

4.2 Résultats et analyses

3. Les sources de ces transformations sont disponibles sur `https://github.com/TurpIF/tp-markov-chain`

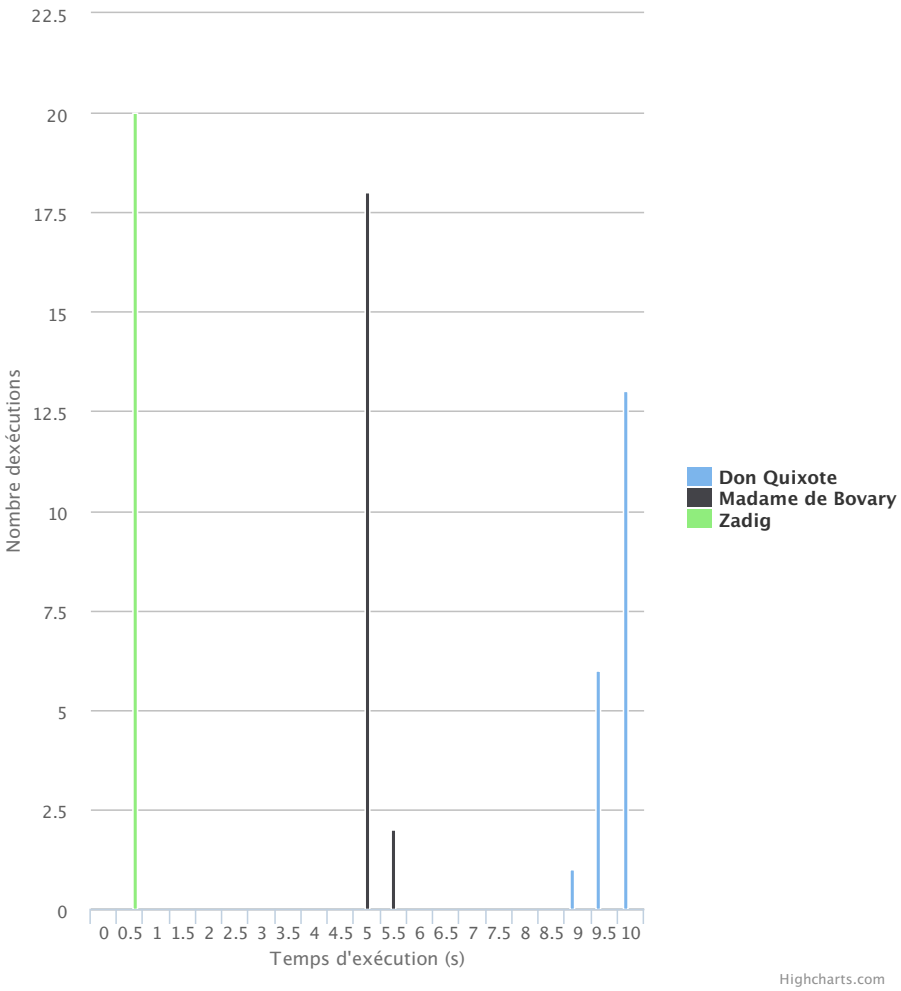

FIGURE 2 – Temps d'exécution du programme avec k=3 et m=1000000 avec toute les exécutions en parallèles

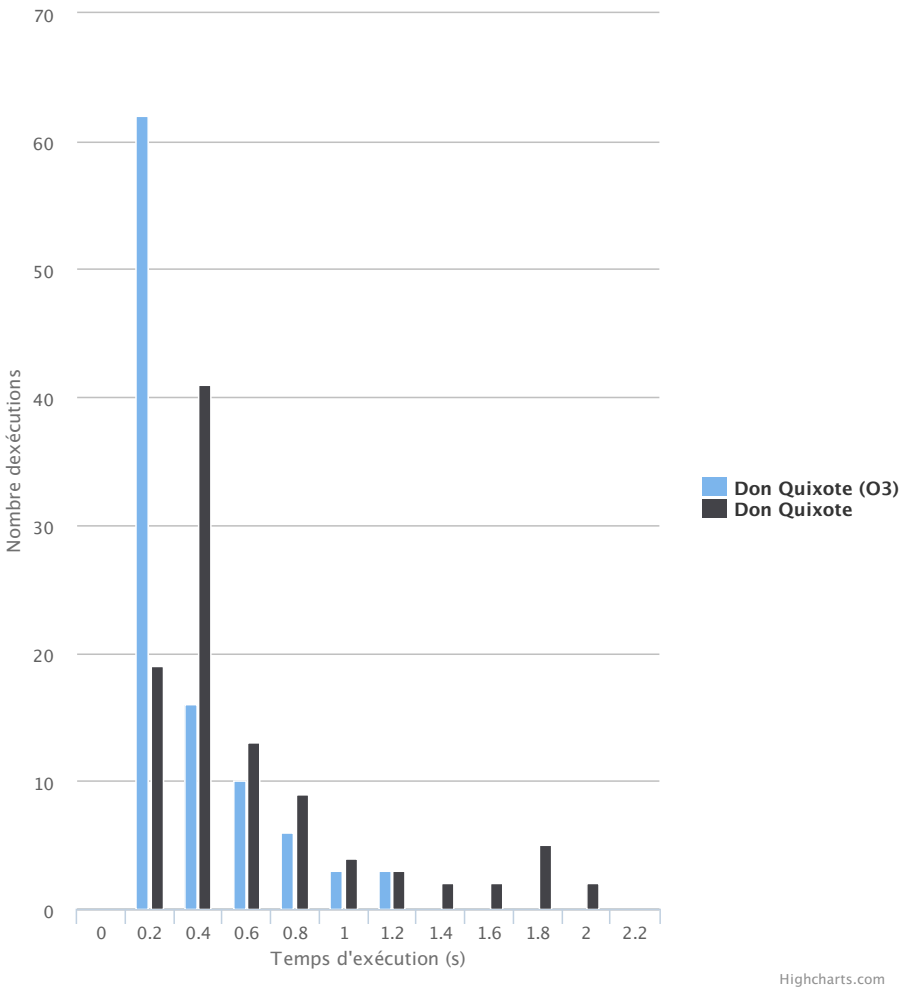

FIGURE 3 – Comparaison des temps d'exécution du programme avec k=3 et m=1000000 avec optimisation et sans