Codage de Prüfer

Maxime GOFFART - Olivier JORIS

2019 - 2020

Table des matières

1 Introduction

Le codage de Prüfer permet d'encoder un arbre composé de n sommets (avec des labels différents compris entre 1 et n) via une suite de n - 2 naturels. Il existe donc une bijection entre un arbre dont les n sommets sont numérotés et son codage de Prüfer, ce qui implique qu'un seul arbre admet un seul codage de Prüfer et inversement.
Cette bijection permet de facilement démontrer la formule de Cayley :
Soit $n > 1$, on peut construire exactement n^{n-2} arbres différents constitués de n sommets.

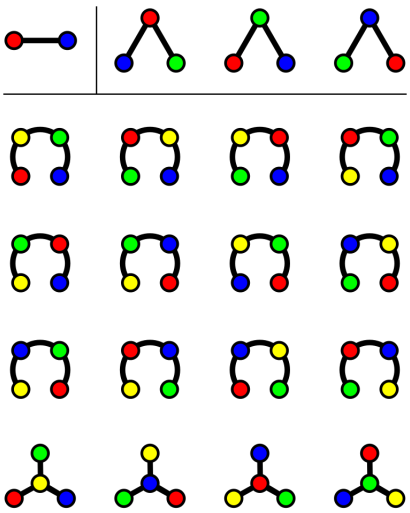

FIGURE 1 – Liste complète des arbres à 2, 3 et 4 sommets [1]

Ce codage permet également de représenter un arbre sans utiliser sa matrice d'adjacence ou sa liste d'adjacence. De cela découle une optimisation de sa complexité spatiale et par conséquent des complexités temporelles des algorithmes le manipulant.

Nous avons réalisé un programme qui, à un arbre donné, donne son codage de Prüfer et inversement.

2 Mode d'emploi du programme

Le programme peut être compilé à l'aide du *makefile* via la commande suivante :

```
$ make prufer
```

Un exécutable nommé `prufer` est alors créé, celui-ci peut être lancé à l'aide de deux arguments :

```
$ ./prufer -m encodage -f exempleGraphe.txt
```

où l'option `-m` désigne le mode d'utilisation du programme [2]. L'option `-f` désigne le fichier contenant la matrice d'adjacence du graphe en cas d'encodage et la suite de Prüfer (précédée de sa longueur sur la première ligne) en cas de décodage. Des exemples de fichiers sont disponibles dans le dossier `/code`.

1. Source : `https://fr.wikipedia.org/wiki/Formule_de_Cayley`
2. encodage ou decodage

3 Stratégie adoptée

Le codage de Prüfer s'appliquant uniquement aux arbres, nous avons d'abord implémenté des fonctions vérifiant que le graphe fourni par l'utilisateur soit un arbre [3].

Ensuite, nous nous sommes renseignés sur le fonctionnement des algorithmes d'encodage et de décodage (via ce lien : `https://fr.wikipedia.org/wiki/Codage_de_Pr%C3%BCfer`) et nous les avons implémentés.

4 Choix opérés

Nous avons choisi de représenter un codage de Prüfer grâce à la structure suivante, permettant de directement accéder à la taille de la séquence de nombres :

```
typedef struct CodagePrufer_t{
   unsigned int taille; // La taille du codage de Prüfer
   int* suitePrufer; // La suite de Prüfer (suite de nombres)
}CodagePrufer;
```

En ce qui concerne la représentation des graphes, nous avons choisi d'utiliser l'interface `graphes.c/graphes.h` qui nous était proposée en y ajoutant différentes fonctions.

5 Principales fonctions

Les principales fonctions composant notre programme sont :

- Des fonctions vérifiant si le graphe est un arbre :
  - `test_connexite` : Vérifie si un graphe donné en argument est connexe.
  - `est_non_oriente` : Vérifie si un graphe donné en argument est non orienté.
  - `contient_cycle` : Vérifie si un graphe donné en argument contient au moins un cycle.
- Des fonctions d'encodage d'un graphe et de décodage d'une séquence de Prüfer :
  - `generer_codage_prufer` : Génère le codage associé à l'arbre donné.
  - `decoder_codage_prufer` : Génère l'arbre associé au codage de Prüfer donné.

6 Difficultés rencontrées

Les principales difficultés rencontrées étaient liées aux manipulations de liste d'adjacence du graphe donné. Un schéma représentant celle-ci facilite leur manipulation.

7 Résultats expérimentaux

Les résultats suivants ont été obtenus en mesurant empiriquement les temps de calcul des fonctions d'encodage d'un arbre et de décodage d'une suite de Prüfer. Ceux-ci ont été obtenus en générant aléatoirement une suite de Prüfer qui a ensuite été décodée puis, grâce au graphe résultant, a été encodée. Les fonctions prévues à cet effet sont situées dans le module

3. En cas d'encodage

`prufer_benchmarking.c`. Ce module peut être compilé séparément du programme principal grâce à l'utilisation de la commande suivante :

```
$ make prufer_benchmarking
```

Un exécutable `prufer_benchmarking` est alors créé, celui-ci peut-être lancé à l'aide d'un argument désignant le nombre de sommets composant l'arbre généré aléatoirement :

```
$ ./prufer_benchmarking 1000
```

Une fois le temps de calcul écoulé, celui-ci est disponible avec le nombre de sommet correspondant sur une nouvelle ligne des fichiers `resultats_encodage.txt` pour le temps d'encodage et `resultats_decodage.txt` pour le temps de décodage.

Ces tests ont été effectués sur une machine dotée d'un Intel Core i5 de 2 coeurs cadencés à 2.3 GHz. Le code a été compilé avec l'optimisation au niveau [4] 3.

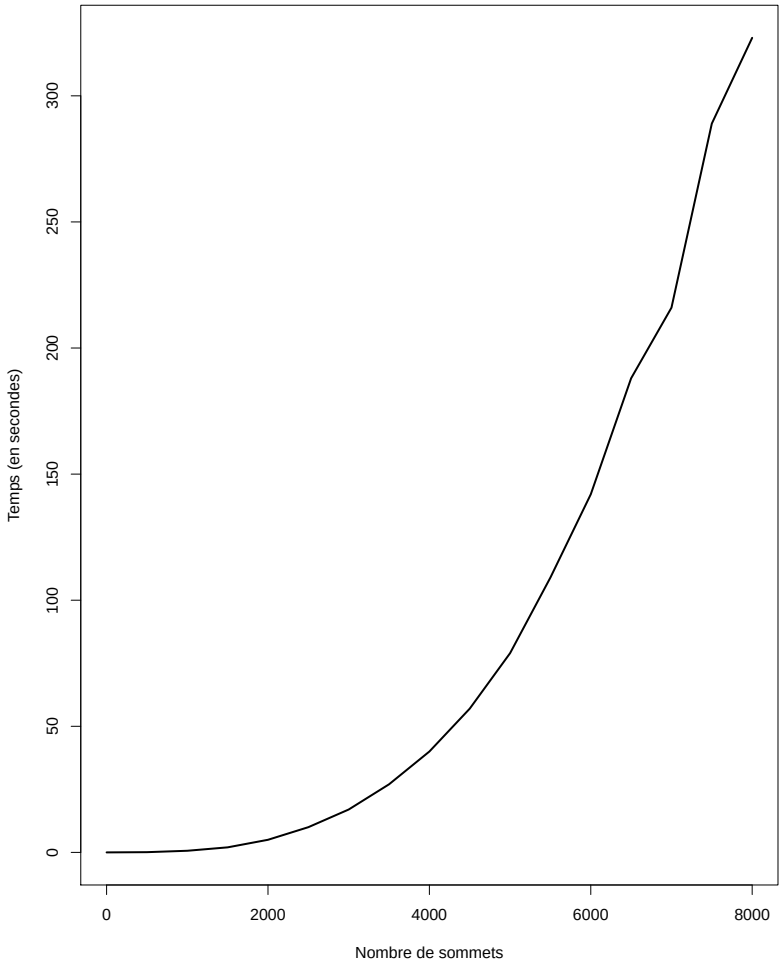

FIGURE 2 – Temps d'encodage en fonction du nombre de sommets de l'arbre

4. L'option `-O3` de `GCC`.

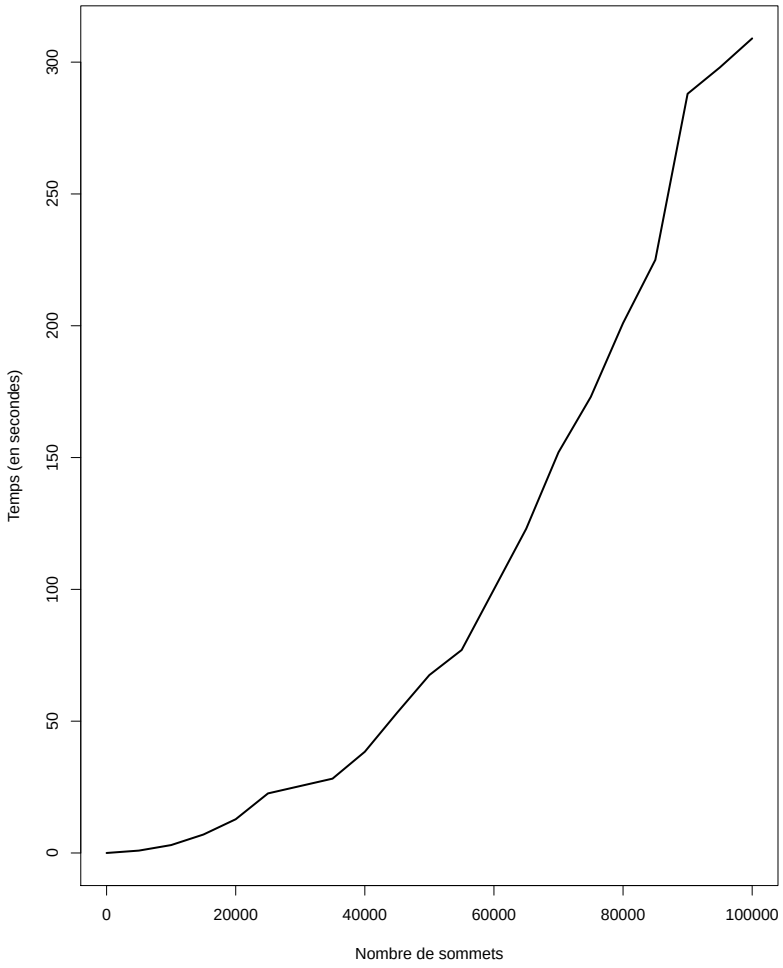

FIGURE 3 – Temps de décodage en fonction du nombre de sommets de l'arbre

Sur base des figure 2 et 3, on voit que le temps d'encodage est nettement plus élevé que le temps de décodage. En effet, contrairement au décodage, l'encodage nécessite de rechercher la feuille ayant le plus petit label, ce qui est gourmand en temps.