PROJET SE

Storck Jefferson

2025

Introduction

Le but de ce projet est d'étudier différents modes de compression pour rendre la transmission de tableaus d'entier plus efficace. Dans ce rapport nous approfondirons sur ces modes, les problèmes rencontraient lors de leur implémentation ainsi que les solutions trouvées pour les réaliser. Nous nous pencherons aussi sur l'implémentation générale de l'application et la mesure de son efficacité.

Mode de compression

Il existe trois modes de compression: **overlap**, **non-overlap** et **overflow**. Nous verrons chacun d'entre eux.

2.1 Implémentation generale

Les modes de compression ont été implémentés autour d'une classe abstraite nommée **Type**, cette classe fournit les méthodes abstraites **compress**, **decompress** et **get** qui sont ensuite surchargées par les classes qui en hérite. Elle fournit aussi la méthode **count**, dont le but est de trouver le nombre de bits necessaires à la représentation binaire du plus grand entier de la table. Les methodes **padding** et **reversePadding** sont aussi héritées de cette classe, nous les approfondirons plus tard. Le programme ne gère pas les nombres négatifs, il gère cependant la range complète des entiers non-signés, pour ce faire le programme manipule des **Long** et non des **Int**.

```
1  protected int count(List<Long> t) {
2    int res=0;
3    for(long i:t) {
4      if(Long.toBinaryString(i).length()>res) {
5  res=Long.toBinaryString(i).length();
6      }
7    }
8    return res;
9  }
```

Chaque mode de compression a une classe qui lui est propre et hérite de **Type**. Le programme crée une nouvelle instance de ces classes à l'aide de la classe **Factory** qui est une implémentation du patron de conception fabrique, cette classe possède une méthode statique **build** qui en fonction de l'argument donné en paramètre va renvoyer une instance de la classe demandée.

2.2 Overlap

overlap est implémenté par la classe **Over** qui hérite de **Type**.

2.2.1 Compression et decompression

Un des principaux problèmes rencontrés lors de l'implémentation est le conservation des bits zéro entre la compression et décompression, en effet la façon dont l'**Overlap** est implémenté implique qu'il est possible que la représentation binaire d'un nombre après compression sur 32 bits commence par un nombre de zéros donnés qui sont perdus lors de la convertion du nombre vers sa reprèsentation binaire. Pour y remédier nous avons la méthode **padding** héritée de **Type**.

```
1  protected String padding(String s,int n) {
2    String res=s;
3    while(res.length()<n) {
4      res="0"+res;
5    }
6    return res;
7  }
```

Cette méthode rajoute à la représentation binaire les zéros perdus. La décompression nécessite de connaître la longueur standard utilisée lors de la compression, cette information est demandée à l'utilisateur.

Un autre problème rencontré est celui de la représentation binaire du dernier nombre du tableau compressé, en effet ici la méthode **padding** n'est pas addaptée car nous ne pouvons pas différentier les bits de "paddings" et les bits réels lors de la décompression, nous utilisons donc la méthode **reversePadding** à la place. Cette méthode fonctionne comme **padding** à l'exception qu'elle rajoute les zéros à droite de la représentation, elle change donc la valeur du nombre. L'utilisation de cette méthode crée un nouveau problème cependant, elle depend du fait que les tableaux à compressés ne contiennent pas de valeurs nulles(zéro).

2.2.2 Get

La méthode **get** fonctionne de façon similaire à la decompression, l'indice de la représentation binaire contenant le nombre demandé est obtenu à l'aide de la formule suivante:

$$indice = l * (i - 1)$$

où **l** est la longeur standard et **i** l'indice du nombre demandé.

2.3 Non-Overlap

non-overlap est implémenté par la classe **NoOver** qui hérite de **Type**.

2.3.1 Compression et decompression

Ce mode fonctionne de façon similaire au traitement du dernier nombre de **Overlap**, on utilise **reversePadding** pour combler les représentations binaires des nombres compressés finaux.

2.3.2 Get

La méthode **get** fonctionne de façon similaire à celle de **overlap**

$$indice = l * (i - 1)$$

où **l** est la longeur standard et **i** l'indice du nombre demandé.

2.4 Overflow

L'**overflow** est implémenté par la classe **Flow**, il n'est cependant pas fonctionnel.

2.4.1 Compression et decompression

Le principal problème a été rencontré durant la décompression, comment différencier les entiers compressés de la partie overflow. Je n'ai pas trouvé de moyen efficace de le faire sans demander à l'utilisateur des infomations suplèmentaire à passer en paramètre, comme le nombre d'entiers overflow et leur longueur standard.

Mesure

Le programme mesure le temps pris par chaque fonction à leur exécution comme suit:

```
1  long start=System.nanoTime();
2  bit.compress();
3  long end=System.nanoTime();
4  System.out.print("Time elapsed :");
5  System.out.println(Math.round(end-start/1000000.0)+"ms");
```

A l'aide de cette méthode on pourrait imaginer un protocol pour déterminer quand la compression est interéssante. Par exemple on pourrait créer une fonction qui génére des tableaux d'entiers aléatoire à longeur variable et exécute une compression et décompression sur plusieurs centaines d'entre eux dans le but d'établir une moyenne du temps pris par chaque fonction. En se basant sur ces moyenne on pourrait determiner quand il est interessant ou non de compresser avant une transmission de latence t.

Conclusion

Si l'implémentation de la compression pour le mode **overlap** et **non-overlap** fonctionne, le façon dont le decompression et le **get** on été implémentés nécessite de connaître à l'avance la longueur standard utilisée durant la compression. Le mode **flow** quant à lui ne dispose pas d'une implémentation satisfaisante.