Rapport de Projet : Compression d'entiers

Bit Packing :

Auteur : ALECU Luca

Enseignant : Jean-Charles Régin

Matière : Software Engeenering

Sommaire :

1. Introduction :

L'objectif de ce projet est de simplifier la transmission de données en se concentrant sur la compression de tableaux d'entiers très souvent utilisés en informatique. Le principe de compression abordé par ce projet est celui de la compression binaire appelée Bit Packing dont le fonctionnement consiste à réduire l'espace mémoire nécessaire pour stocker un tableau d'entiers tout en conservant la possibilité d'accéder rapidement à un élément donné sans devoir décompresser l'ensemble des données. Trois versions distinctes de l'algorithme ont été implémentées afin de pouvoir comparer leurs efficacités et leurs performances.

2. Présentation de la solution :

La solution repose sur une architecture orientée objet, composée de plusieurs classes interconnectées :

Bitpacking : une classe abstraite dont les 2 versions principales Bitpackingcross et Bitpackingnocross hériterons, elle définit les méthodes de base (compress, decompress et get) et contient des méthode dont toutes les versions auront besoin (bitsNeeded et measure).

Bitpackingcross : une classe qui implémente la compression où les entiers peuvent franchir les frontières entre deux entiers 32 bits.

Bitpackingnocross : une classe qui implémente la version plus stricte, sans chevauchement, garantissant qu'un entier compressé reste contenu dans un seul mot de 32 bits.

Bitpackingoverflow : une classe qui hérite de Bitpackingcross afin de pouvoir utiliser ces méthodes dans certaines étapes de la compression améliore cette même compression en isolant les valeurs nécessitant beaucoup de bits.

Bitpackingfactory : une classe implémentant le design pattern factory qui permet de créer dynamiquement une instance du type de compression souhaité.

Benchmark : une classe utilitaire qui permet de faire l'affichage du programme en utilisant toutes les méthodes définies plus tôt et qui test l'efficacité ainsi que la justesse et la robustesse du code.

3. Difficultés d'implémentation :

La première difficulté a été de gérer la compression en se servant des opérateurs de décalage, car j'ai d'abord tenté de le faire sans ces opérateurs mais les opérations devenait plus complexe que nécessaire et me menaient a faire des erreurs de décalage et de masque.

Ensuite la gestion des frontières binaires et des masques: lors de cette partie j'ai même après avoir utilisé des opérateurs de décalages eu des erreurs de compression que j'ai réglé par la suite provoqués par les conversion en long faites au cours du programme.

Dans ma solution je n'ai pas pris en compte les entiers négatifs donc dans le cas d'un tableau contenant des entiers négatifs le code plantera très probablement.

4. Fonctionnement des classes techniques :

4.1 La classe Bitpacking.java :

• 4.1.1 Objectif :

Cette classe abstraite définit la structure commune à toutes les implémentations de Bit Packing. Elle permet de définir les attributs de base des classes de bitpacking qui sont le tableau compressé par la méthode, le nombre d'éléments d'origines et le nombre de bits nécessaire pour coder un entier du tableau ; De même elle contient les méthodes utilitaires pour déterminer le nombre de bits nécessaires à la représentation des valeurs et pour mesurer les performances d'exécution.

• 4.1.2 bitsNeeded(int[] array) :

Cette méthode calcule le nombre minimal de bits k nécessaires pour représenter les valeurs d'un tableau. Elle parcourt toutes les valeurs, détermine la valeur maximale, puis applique :

k = 32 - Integer.numberOfLeadingZeros(max);

Ainsi, si la plus grande valeur est 1023 (en binaire 1111111111), on obtient k = 10.

• 4.1.3 measure(Runnable task, int runs) :

Cette fonction mesure les performance des compressions et décompressions en utilisant des Runnable comme paramètre représentant des parts de code a exécuter que j'appelle dans le code a l'aide de lambda sous la forme : () -> bp.compress(data) et runs représente le nombres de fois que le code sera évalué on prendra le meilleur des runs a chaque fois .

4.2 La classe Bitpackingcross.java :

• 4.2.1 Objectif :

Implémenter la version de la compression où les entiers peuvent être écrits sur deux mots consécutifs de 32 bits. Cette approche maximise la compacité du flux binaire, au prix d'une complexité accrue.

• 4.2.2 Méthode compress(int[] array) :

Calcule k le nombre de bits nécessaires pour coder le plus grand entier du tableau pour connaitre la taille du tableau compressé des sont initialisation n*k. Pour chaque valeur, calcule la position du premier bit (bitPos) et écrit la valeur dans le mot correspondant (wordIdx et offset). Si la valeur dépasse la frontière de 32 bits, elle est coupée et la partie restante est écrite dans le mot suivant.

Ce mécanisme garantit une densité maximale du stockage binaire.

• 4.2.3 Méthode decompress(int[] dest) :

Inverse le processus de compression : Lit les k bits de chaque valeur en recomposant les éventuelles parties réparties sur deux mots pour restaurer le tableau original.

• 4.2.4 Méthode get(int i) :

La méthode get(int i) permet de récupérer directement la valeur de l'élément d'indice i dans le tableau compressé, sans décompresser l'ensemble du tableau. Pour accéder à l'élément numéro i, la méthode effectue plusieurs étapes, d'abord le calcul de la position en bits de l'élément dans le flux compressé : bitPos=i*k où k est le nombre de bits utilisés pour représenter chaque entier. Ensuite la détermination du mot mémoire contenant le début de l'entier : wordIdx=bitPos/32. Puis le calcul du décalage (offset) à l'intérieur de ce mot : offset=bitPos%32 (Ce décalage correspond au nombre de bits inutiles précédant la valeur recherchée). Et enfin l'extraction de la valeur : si la valeur tient entièrement dans le mot courant (offset + k ≤ 32), on applique simplement un décalage à droite et un masquage, sinon si la valeur dépasse la limite du mot, on récupère la partie restante dans le mot suivant, puis on combine les deux segments avec un décalage gauche et un OU binaire. La Complexité de cette méthode est constante : O(1).

4.3 La classe Bitpackingnocross.java :

• 4.3.1 Objectif :

Cette classe propose une implémentation plus simple et plus rapide du Bit Packing, au prix d'une utilisation légèrement moins optimale de la mémoire. Le principe est de garantir qu'aucun entier ne soit divisé entre deux mots mémoire de 32 bits. Chaque mot contient plusieurs valeurs consécutives, chacune occupant k bits, mais on ne franchit jamais la frontière d'un mot.

• 4.3.2 Méthode compress(int[] array) :

Calcule k le nombre de bits nécessaires pour coder le plus grand entier du tableau pour connaitre la taille du tableau compressé des sont initialisation, Détermine combien d'éléments peuvent tenir dans un mot (perWord = 32 / k). Insère les valeurs successivement dans le mot, à des positions multiples de k.

• 4.3.3 Méthode decompress(int[] dest) :

Récupère pour chaque élément i la valeur correspondante dans le mot approprié en effectuant l'opération inverse de compress. Pour chaque index i on détermine le mot d'origine : wordIdx=i/perWord, puis on calcule le décalage : offset=(i%perWord)×k et enfin on extrait la valeur et on la range dans le tableau de sortie dest.

• 4.3.4 Méthode get(int i) :

Cette méthode reprend le même principe que la décompression, mais pour un seul élément avec encore une fois un Accès ultra-rapide en O(1) constant.

4.4 La classe Bitpackingoverflow.java :

• 4.4.1 Objectif :

Le but de la classe Bitpackingoverflow est d'améliorer la compression dans les cas où certaines valeurs isolées nécessitent beaucoup plus de bits que les autres. Plutôt que de fixer un k élevé pour tous les éléments, on sépare les valeurs "anormales" dans une zone de débordement (overflow area) et on les référence depuis le flux principal.

• 4.4.2 Choix :

Il est possible d'implémenter cette méthode en réutilisant soit le concept de Bitpackingcross soit Bitpackingnocross et j'ai choisi de l'implémenter avec Bitpackingcross car comme j'ai pu observer que cette version était plus efficace que l'autre, c'est donc pour cela que la classe Bitpackingoverflow hérite de Bitpackingcross et non de Bitpacking.

• 4.4.3 Étapes de la compression :

Tout d'abord j'ai également choisit de déterminer une taille de base k' (souvent la moitié ou les trois quart du k global) ainsi que de réserver 1 bit de drapeau pour indiquer si une valeur appartient à la zone de débordement. Si le flag = 0 : valeur normale alors que si le flag = 1 : référence vers la zone de débordement. De plus j'utilise une structure de données (List<Integer> overflowValues) pour stocker ces grandes valeurs. Ensuite j'identifie les valeurs en débordement : Toute valeur ≥ 2^k' est déplacée dans la zone de débordement. Je garde son index dans une table overflowIndex. Ensuite j'encode le flux principal comme mentionné plus haut et c'est a ce flux encodé qu'on applique la méthode de compression de Bitpackingcross.

• 4.4.4 Décompression des données :

La décompression du flux principal ce fait via super.decompress(tmp).

Ensuite il faut décoder chaque élément :

Si le bit de drapeau = 0 : valeur directe (lecture des k' bits).

Si le bit de drapeau = 1 : récupération de la valeur correspondante dans overflowValues.

On reconstruit ensuite le tableau original.

• 4.4.5 Accès direct à un élément compressé :

Lorsqu'on appelle get(i) :

Je lis l'élément compressé à la position i du flux principal et je sépare le bit de drapeau (bit le plus significatif).

Si flag = 0, on renvoie directement la valeur décodée sinon on récupère dans la liste overflowValues la valeur pointée par l'index.

Ce mécanisme maintient donc encore et toujours un accès direct constant (O(1)).

5. Classe Benchmark et Lancement:

5.1 Lancement :

Dans la classe lancement.java qui est mon main j'ai choisit d'implémenter pour la partie front end que si on lui passe un fichier contenant un tableau en paramètre il récupère son contenu et lance le programme avec ce tableau pour facilité les tests mais également que si on exécute le programme sans rien en paramètre il s'exécutera sur mon Benchmark qui est un tableau de 10000 entiers contenus entre 0 et 5000.

5.2 Choix :

Dans la classe Benchmark.java j'ai effectué l'appel de toutes les mesures et vérification que j'ai jugé nécessaires qui sont donc les temps d'exécution de la compression de la décompression ainsi que du get de certains éléments ainsi que le ratio de compression réussi par l'algorithme qui s'il est égal a 50% par exemple signifie qu'on a divisé l'espace pris par le tableau par 2 ainsi plus ce score est bas plus la compression a été efficace, dans le même esprit j'ai également décidé d'afficher les saved Bytes qui sont le nombre de bit libérés par la compression. Attention cependant toutes ces informations ne constituent en rien l'entièreté des paramètres que j'ai évalué, pour mon usage personnel lors des tests effectués affin de vérifier de façon plus sure la justesse du code notamment celle de la fonction get j'ai utilisé une partie du code que j'ai laissé on commentaire si jamais vous voudriez la tester mais je ne l'ai pas laissée en temps que code pur car maintenant que je suis sur que les résultats sont justes cette partie a perdu en pertinence :

```
47          //System.out.println("Test de justesse decompression:");
48          //for (int i = 0; i < Math.min(10, data.length); i++) {
49          //    System.out.printf(" - indice %d : attendu %d, obtenu %d%n",
50          //                i, data[i], out[i]);
51          //}
52          //System.out.println("Test de justesse get:");
53          //for (int i = 0; i < Math.min(10, data.length); i++) {
54          //    System.out.printf(" - indice %d : attendu %d, obtenu %d%n",
55          //                i, data[i], bp.get(i));
56          //}
```

5.3 Mesures :

Le ratio est obtenu en comparant la taille du tableau compressé à la taille originale de la façon suivante : ((taille compressée)/(taille originale))*100

Les trois variantes (Cross, NoCross, Overflow) sont comparées sur les mêmes données pour observer les compromis entre :

Rapidité de compression ,Rapidité de décompression ,Taille compressée ,Temps d'accès direct.

Les mesures permettent de tirer plusieurs enseignements :

BitPackingCross offre la meilleure réduction de taille, mais la compression et la décompression sont légèrement plus longues en raison de la gestion des chevauchements.

BitPackingNoCross est plus rapide, car les opérations bit à bit sont localisées dans un seul mot mémoire.

BitPackingOverflow est plus efficace sur des données hétérogènes, car elle évite de gaspiller des bits pour des valeurs extrêmes.

L'accès direct via get(i) est quasi instantané (O(1)) dans tous les cas.

5.4 Vérifications :

Il est facile de vérifier si le code fonctionne ou non c'est d'ailleurs à ça que servent la variable booléenne vérification et l'affichage vérification globale. L'intérêt sur ce point ce porte plutôt sur les tests effectués sur ce programme.

6. Tests :

De très nombreux tests aléatoires de très grande taille ont été produits par le benchmark ainsi qu'une batterie de tests unitaire particuliers comme un tableau ne contenant que des 0 ou un tableau contenant des valeurs très éloignées ect.

(Afin que les tests fonctionnent le fichier test.txt passé en paramètre doit ce trouver dans le même dossier que le main).