

Cycle ingénieur - 2ème année GSI

Programmation fonctionnelle

Projet *Méthodes de compression sans perte*

2023-2024

Consignes

Objectif du projet

- Implémenter certaines méthodes de compression sans perte

Consignes générales

- Projet effectué par groupes de **4 à 5 étudiants**
- Date limite de rendu (sur CoursCY) : **31 mars 2024, 23h59**
- Soutenance : semaine du **1er au 5 avril 2024**

Nature du rendu

- *Fourni* : code à compléter
- **À faire** : implémenter les éléments manquants dans chaque module
- **Rendu** : archive contenant le répertoire src et rapport court

Modification du code existant

- Modification **INTERDITE** :
 - de la signature des modules
 - de la signature des valeurs et fonctions à implémenter
 - des (quelques) éléments déjà implémentés
- Ajout **INTERDIT** de paquets supplémentaires
- Ajout *autorisé* de nouveaux éléments intermédiaires pour implémenter les éléments demandés

Tests des valeurs et méthodes à implémenter

- Des tests seront effectués sur votre code. **Ces tests participent à la note finale du projet.**
- Pour fonctionner, **ces tests supposent que les consignes précédentes ont été respectées.** *Dans le cas contraire, les tests ne fonctionneront pas.*

Vous pouvez créer vos propres tests (stack test) !

- HUnit : inspiré de JUnit
- QuickCheck : orienté *property testing*
- HSpec : orienté *comportement*, gère l'intégration des deux précédentes

Consignes spécifiques

Respect des règles de programmation fonctionnelle

- Utilisation de variables **interdite**
- Utilisation des boucles (conditionnelles ou non) **interdite**

Quelques conseils

- **Réduire le recours à la récursivité** en utilisant les fonctions :
 - du paquetage standard
 - du paquetage containers
- Sinon **favoriser la récursivité terminale**

Rapport de développement

- Description succincte de la conception mise en place
- Description succincte de l'implémentation mise en place

Étude des méthodes de compression

- Méthode RLE
 - quelles configurations donnent les meilleures performances ?
- Méthodes statistiques :
 - comparaison Huffman / Shannon-Fano
 - quelles configurations donnent les meilleures performances ?
- Méthodes à dictionnaire :
 - comparaison LZ78 / LZW
 - quelles configurations donnent les meilleures performances ?

Soutenance (semaine du 1/4 au 5/4)

Organisation

- Durée totale : *20 minutes*
 - Temps de présentation : **entre 10 et 15 min**
Contrainte de temps à respecter absolument
 - Reste du temps consacré à nos questions
- **Tous les membres du groupe doivent présenter.**

Attendu

- Présenter la solution implémentée
- **Présenter un jeu de test pour valider l'implémentation**

Contenu du répertoire src fourni

Aucune modification n'est requise dans les fichiers en *italique*.

- src
 - RLE.hs
 - Statistic
 - Bit.hs*
 - EncodingTree.hs
 - Huffman.hs
 - ShannonFano.hs
 - Source.hs
 - LZ
 - Dictionaries.hs*
 - LZ78.hs
 - LZW.hs

Présentation du projet

Présentation du projet

Présentation générale

Motivation de la compression sans perte

- Réduire le temps de transfert en réduisant la taille
⇒ algorithme de compression
- Résultat de l'algorithme souvent peu lisible
⇒ algorithme de décompression
- Compression **sans perte** : la décompression doit reconstruire les données originelles **à l'identique**

Méthodes étudiées dans ce projet

- RLE (*Run-Length Encoding*)
- Méthodes statistiques : Huffman, Shannon-Fano
- Méthodes à dictionnaire : LZ78, LZW

Données manipulées

- Liste de symboles
- a : type d'un symbole
- *Cas usuel : chaînes de caractères* i.e. $\text{String} \equiv [\text{Char}] \Rightarrow a = \text{Char}$
- Résultat de la compression : *dépend fortement de la méthode utilisée*

Fonctions pour toute méthode de compression

Une méthode de compression définit donc deux opérations :

- `compress` transforme la liste de symboles en un objet compressé
- `uncompress` effectue (si possible $\leadsto \text{Maybe } _$) l'opération inverse pour retrouver la liste originale **à l'identique**

Autrement dit : **`uncompress (compress msg) == (Just msg)`**

Présentation du projet

RLE

La méthode consiste à « compresser » les symboles identiques qui se suivent.

Exemples

- "" → []
- "a" → [(a, 1)]
- "aaaabbbcbbbb" → [(a, 4), (b, 2), (c, 1), (b, 3)]

À implémenter dans `src/RLE.hs`

- `compress` : méthode de compression RLE
- `uncompress` : méthode de décompression RLE

Présentation du projet

Méthodes statistiques

Ces méthodes s'appuyant sur la connaissance de la distribution des symboles dans le message à compresser (appelée *source*).

On commence donc par calculer cette distribution et quelques indicateurs.

À implémenter dans `src/Statistic/Source.hs`

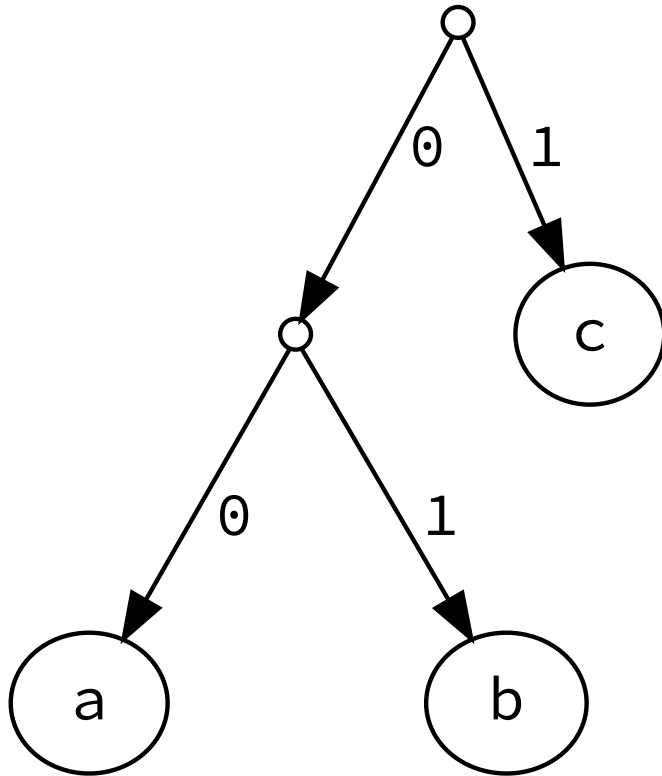
- `occurrences` : tableau associatif donnant, pour chaque symbole présent, le nombre de fois qu'il apparaît dans la source
- `entropy` : l'entropie de la source, i.e.
$$\text{entropy} = - \sum_s \frac{n(s)}{N} \cdot \log_2 \left(\frac{n(s)}{N} \right)$$
 - N : longueur de la source
 - $n(s)$: nombre d'apparitions du symbole s dans la source
- `orderedCounts` : liste des couples (symbole, effectif) triée par ordre croissant du nombre d'apparitions dans la source

Résultat de la compression

- Chaque symbole est représenté par une suite de bits
⇒ résultat : liste de bits
- Type Bit fourni (src/Statistic/Bit.hs) avec :
 - la constante Zero (pour 0)
 - la constante One (pour 1)

Représentation du codage binaire

- Chaque méthode définit donc un codage binaire de chaque symbole.
- Un tel codage est représenté sous la forme d'un arbre binaire dont les feuilles sont les symboles apparaissant dans la source.



<i>Symbole</i>	'a'	'b'	'c'
<i>Codage binaire</i>	00	01	1

- "a" → 00
- "ab" → 0001
- "abbca" → 000101100

La convention utilisée pour identifier les branches est totalement arbitraire et ne nuit à la performance.

Type EncodingTree a

Description

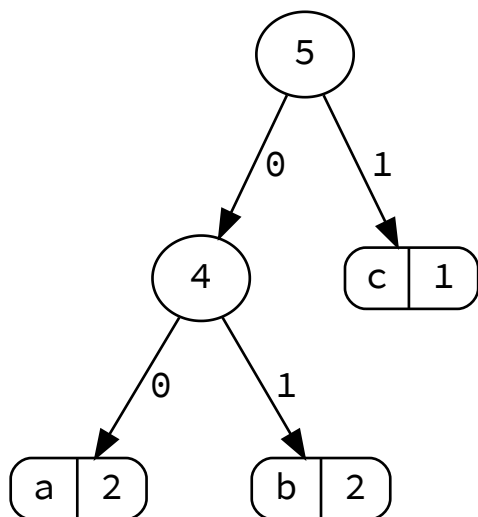
Arbre binaire *étiqueté* par des entiers :

- Feuille : `EncodingLeaf lbl v`
 - `lbl : Int → étiquette ;`
 - `v : a → valeur de la feuille (symbole)`
- Nœud : `EncodingNode lbl l r`
 - `lbl : Int → étiquette ;`
 - `l, r : EncodingTree a → sous-arbres gauche et droit`

Quelle valeur pour les étiquettes ?

- Feuille : nombre d'occurrences du symbole dans la source
- Nœud : somme des étiquettes des sous-arbres

Exemple avec la source "abbca" (entropie $\approx 1,52$)



Symbole	'a'	'b'	'c'
Codage binaire	00	01	1

Longueur moyenne du code :

$$\frac{2}{5} \cdot 2 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 1 = \frac{9}{5} = \mathbf{1,8}$$

Remarques

- D'autres arbres existent (avec des longueurs moyennes différentes).
- **Plus la longueur moyenne est courte, plus le codage est efficace.**
- Limite inférieure théorique *sans perte* : **entropie de la source**

Type EncodingTree a

À implémenter dans `src/Statistic/EncodingTree.hs`

- `has` : teste si un symbole se trouve dans une feuille de l'arbre
- `meanLength` : calcule la longueur moyenne du code associée à l'arbre
- `encode` : transforme un symbole en son code binaire
- `decodeOnce` : transforme (si possible) les premiers bits en un symbole
- `decode` : transforme (si possible) la liste de bits en une liste de symboles
ATTENTION : le décodage doit tomber juste avec le dernier bit
- `compress` : encode la liste de symboles en une liste de bits à partir de la fonction générant l'arbre d'encodage
- `uncompress` : décode (si possible) la liste de bits pour retrouver la liste de symboles originale

Principe de construction de l'arbre de codage

1. Trier la distribution des symboles par ordre décroissant du nombre d'apparitions
2. Fusionner les deux dernières colonnes et les réinsérer en conservant l'ordre décroissant
3. Recommencer l'étape 2 jusqu'à obtenir une seule colonne
4. Construire l'arbre en inversant les fusions successives

À implémenter dans `src/Statistic/Huffman.hs`

- `tree` : calcule l'arbre d'encodage selon cette méthode

Exemple avec la source "abbca" (entropie $\approx 1,52$)

1.

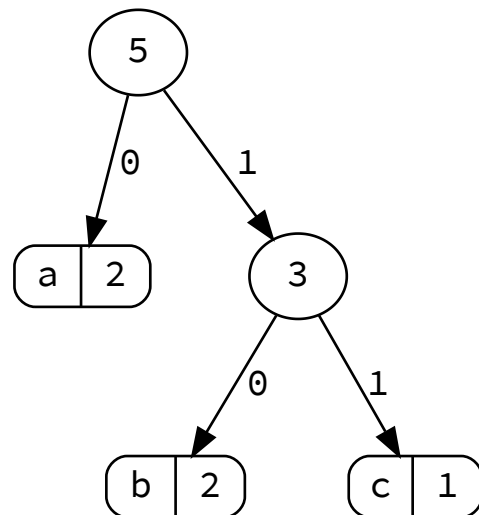
<i>Symbole</i>	a	b	c
<i>Effectif</i>	2	2	1

2.

<i>Symbole</i>	bc	a
<i>Effectif</i>	3	2

3.

<i>Symbole</i>	abc
<i>Effectif</i>	5



a	b	c
0	10	11

Longueur moyenne :

$$\frac{2}{5} \cdot 1 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 2 = \frac{8}{5} = \mathbf{1,6}$$

Arbre de codage selon la méthode de Shannon-Fano

Principe de construction de l'arbre de codage

1. Trier la distribution des symboles rangée par ordre décroissant du nombre d'apparitions
2. Couper en deux sous-distributions *les plus équilibrées possibles*
3. Recommencer l'étape 2 sur chaque sous-distribution jusqu'à obtenir des distributions à un symbole
4. Construire l'arbre à partir des coupes

(Les performances ne sont pas meilleures que celles de Huffman.)

À implémenter dans `src/Statistic/ShannonFano.hs`

- `tree` : calcule l'arbre d'encodage selon cette méthode

Arbre de codage selon la méthode de Shannon-Fano

Exemple avec la source "abbca" (entropie $\approx 1,52$)

1.

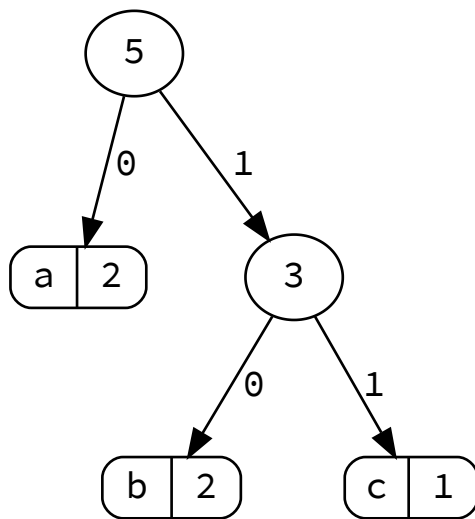
a	b	c
2	2	1

2.

a	b	c
2	2	1

3.

a	b	c
2	2	1



a	b	c
0	10	11

Longueur moyenne :

$$\frac{2}{5} \cdot 1 + \frac{2}{5} \cdot 2 + \frac{1}{5} \cdot 2 = \frac{8}{5} = \mathbf{1,6}$$

Présentation du projet

Méthodes à dictionnaire

Principes généraux

- Encodage des séquences de caractères par des références à des emplacements dans un dictionnaire
- *Dictionnaire construit à partir du texte lui-même* contenant toutes les séquences déjà rencontrées
- Très utilisé dans les logiciels de compression/décompression

Avantages lors de la décompression

- Reconstruction du dictionnaire au fur à mesure \Rightarrow inutile de le fournir
- Code statistique : nécessite l'arbre d'encodage \Rightarrow qualité réelle moindre

Restriction du type de données

- Données manipulées : chaînes de caractères \Rightarrow a = Char
- Résultats de la compression :
 - LZ78 : [(Int, Char)]
 - LZW : [Int]

Objet Dictionaries fourni

- `type Dictionary = [String]`
- Dictionnaires prédéfinis :
 - `empty` : dictionnaire vide (pour LZ78)
 - `ascii` : table ASCII (pour LZW)

Méthode de compression LZ78

Contenu du dictionnaire

1. Initialement : chaîne vide en position 0
2. Recherche de la chaîne maximale existante
3. Ajout au dictionnaire du couple (entier, caractère) avec
 - numéro de la chaîne maximale trouvée
 - dernier caractère suivant cette chaîne

À implémenter dans `src/LZ/LZ78.hs`

- `compress` : méthode de compression LZ78
- `uncompress` : méthode de décompression LZ78

Compression de "belle échelle !"

1. "b" en position 1 du dictionnaire, ajout de (0, 'b')
2. "e" en position 2 du dictionnaire, ajout de (0, 'e')
3. "l" en position 3 du dictionnaire, ajout de (0, 'l')
4. "le" en position 4 du dictionnaire, ajout de (3, 'e')
5. "␣" en position 5 du dictionnaire, ajout de (0, '␣')
6. "ec" en position 6 du dictionnaire, ajout de (2, 'c')
7. "h" en position 7 du dictionnaire, ajout de (0, 'h')
8. "el" en position 8 du dictionnaire, ajout de (2, 'l')
9. "le␣" en position 9 du dictionnaire, ajout de (4, '␣')
10. "!" en position 10 du dictionnaire, ajout de (0, '!')

→ [(0, 'b'), (0, 'e'), (0, 'l'), (3, 'e'), (0, '␣'), (2, 'c'), (0, 'h'), (2, 'l'), (4, '␣'), (0, '!')]

Décompression

1. $(0, 'b')$ = "b" ajouté en position 1 dans le dictionnaire
 2. $(0, 'e')$ = "e" ajouté en position 2 dans le dictionnaire
 3. $(0, 'l')$ = "l" ajouté en position 3 dans le dictionnaire
 4. $(3, 'e')$ = "le" ajouté en position 4 dans le dictionnaire
 5. $(0, '␣')$ = "␣" ajouté en position 5 dans le dictionnaire
 6. $(2, 'c')$ = "ec" ajouté en position 6 dans le dictionnaire
 7. $(0, 'h')$ = "h" ajouté en position 7 dans le dictionnaire
 8. $(2, 'l')$ = "el" ajouté en position 8 dans le dictionnaire
 9. $(4, '␣')$ = "le␣" ajouté en position 9 dans le dictionnaire
 10. $(0, '!')$ = "!" ajouté en position 10 dans le dictionnaire
- "belle echelle !" (**même dictionnaire que pendant la compression**)

Méthode de compression LZW

Variante de la méthode LZ78

- Dictionnaire initial non vide
(*Par défaut, table ASCII de 0 à 255*)
- À l'apparition d'une nouvelle séquence :
 - on ajoute la chaîne maximale au résultat ;
 - on reprend la lecture **depuis le nouveau caractère**.

À implémenter dans `src/LZ/LZW.hs`

- `compress` : méthode de compression LZW
- `uncompress` : méthode de décompression LZW

Compression de "belle echelle" (table ASCII de 0 à 255)

1. "be" en position 256 du dictionnaire, ajout de 98 ('b')
2. "el" en position 257 du dictionnaire, ajout de 101 ('e')
3. "ll" en position 258 du dictionnaire, ajout de 108 ('l')
4. "le" en position 259 du dictionnaire, ajout de 108 ('l')
5. "e_" en position 260 du dictionnaire, ajout de 101 ('e')
6. "_e" en position 261 du dictionnaire, ajout de 32 ('_')
7. "ec" en position 262 du dictionnaire, ajout de 101 ('e')
8. "ch" en position 263 du dictionnaire, ajout de 99 ('c')
9. "he" en position 264 du dictionnaire, ajout de 104 ('h')
10. "ell" en position 265 du dictionnaire, ajout de 257
11. "le", ajout de 259

→ [98, 101, 108, 108, 101, 32, 101, 99, 104, 257, 259]

Décompression

1. 98 ('b') : ajout de "b"
2. 101 ('e') : ajout de "e", "be" en position 256 du dictionnaire
3. 108 ('l') : ajout de "l", "el" en position 257 du dictionnaire
4. 108 ('l') : ajout de "l", "ll" en position 258 du dictionnaire
5. 101 ('e') : ajout de "e", "le" en position 259 du dictionnaire
6. 32 ('_') : ajout de "_", "e_" en position 260 du dictionnaire
7. 101 ('e') : ajout de "e", "_e" en position 261 du dictionnaire
8. 99 ('c') : ajout de "c", "ec" en position 262 du dictionnaire
9. 104 ('h') : ajout de "h", "ch" en position 263 du dictionnaire
10. 257 : ajout de "el", "he" en position 264 du dictionnaire
11. 259 : ajout de "le"

→ "belle echelle" (**même dictionnaire que pendant la compression**)