Compression de texte

Quentin Fortier

April 7, 2022

Code (sale) Tests

Définitions

- Un **alphabet** est un ensemble Σ **fini**, dont les éléments sont des **lettres**.

Définitions

- Un **alphabet** est un ensemble Σ **fini**, dont les éléments sont des **lettres**.
- Un **mot** m d'un alphabet Σ est une suite finie m_1, ..., m_n de lettres de Σ, et on note $m = m_1 ... m_n$.
  n est la **longueur** ($=$ nombre de lettres) de m, qu'on note $|m|$.

Définitions

- Un **alphabet** est un ensemble Σ **fini**, dont les éléments sont des **lettres**.
- Un **mot** m d'un alphabet Σ est une suite finie m_1, ..., m_n de lettres de Σ, et on note $m = m_1 ... m_n$.
  n est la **longueur** ($=$ nombre de lettres) de m, qu'on note $|m|$.
- On note Σ^* l'ensemble des mots de Σ.

Définitions

- Un **alphabet** est un ensemble Σ **fini**, dont les éléments sont des **lettres**.
- Un **mot** m d'un alphabet Σ est une suite finie m_1, ..., m_n de lettres de Σ, et on note $m = m_1 ... m_n$.
  n est la **longueur** (= nombre de lettres) de m, qu'on note $|m|$.
- On note Σ^* l'ensemble des mots de Σ.
- Un mot m_1 est **préfixe** d'un mot m_2 si m_2 commence par m_1.

Exemples :

- Si $\Sigma = \{a, b\}$ alors $ababba$ est un mot de longueur 6. aba est un préfixe de $ababba$.
- Si $\Sigma = \{+, -, /, *, , (,), 0, ..., 9\}$ alors les mots de Σ sont les expressions arithmétiques. Par exemple, $3 * (4 + 5)$ est un mot de longueur 7.

Définitions

Soit Σ un alphabet.

Définition

Un algorithme de compression sans perte consiste à définir deux fonctions $f : \Sigma^* \longrightarrow \Sigma^*$ (codage) et $g : \Sigma^* \longrightarrow \Sigma^*$ (décodage) telles que $f \circ g = id$ et en espérant que $|f(m)|$ soit petit par rapport $|m|$.

Définitions

Soit Σ un alphabet.

Définition

Un algorithme de compression sans perte consiste à définir deux fonctions $f : \Sigma^* \longrightarrow \Sigma^*$ (codage) et $g : \Sigma^* \longrightarrow \Sigma^*$ (décodage) telles que $f \circ g = id$ et en espérant que $|f(m)|$ soit petit par rapport $|m|$.

Ceci définition implique que f est bijective (il existe un unique décodage).

Théorème

Il n'existe pas de fonction injective $f : \Sigma^* \longrightarrow \Sigma^*$ telle que $|f(m)| < |m|$ pour tout $m \in \Sigma^*$.

Définitions

Soit Σ un alphabet.

Définition

Un algorithme de compression sans perte consiste à définir deux fonctions $f : \Sigma^* \longrightarrow \Sigma^*$ (codage) et $g : \Sigma^* \longrightarrow \Sigma^*$ (décodage) telles que $f \circ g = id$ et en espérant que $|f(m)|$ soit petit par rapport $|m|$.

Ceci définition implique que f est bijective (il existe un unique décodage).

Théorème

Il n'existe pas de fonction injective $f : \Sigma^* \longrightarrow \Sigma^*$ telle que $|f(m)| < |m|$ pour tout $m \in \Sigma^*$.

Il n'existe donc pas d'algorithme de compression sans perte qui diminue toujours la taille. Par contre, on peut faire en sorte que ce soit le cas pour un ensemble de mots particuliers (textes en français, par exemple).

Algorithmes de compression

Algorithmes de compression que nous allons voir :

1. Run-Length Encoding (hors programme mais simple)
2. Huffman : algorithme glouton
3. LZW pour Lempel-Ziv-Welch : fenêtre glissante (utilisé par les formats gif et tiff)

Run-Length Encoding (RLE)

Idée : compresser `"aaaabccbbb"` en `[('a', 4); ('b', 1); ('c', 2); ('b', 3)]`.

Exercice

Écrire des fonctions `rle_code : string -> (char*int) list` et `rle_decode : (char*int) list -> string` réalisant cette compression / décompression.

Compression de Huffman

Idée du codage de Huffman : utiliser moins de bits pour les lettres qui apparaissent souvent.

Compression de Huffman

Idée du codage de Huffman : utiliser moins de bits pour les lettres qui apparaissent souvent.

1. Prétraitement : On construit un arbre binaire T qu'on appelle **arbre de Huffman** dont les feuilles sont les lettres.

Compression de Huffman

Idée du codage de Huffman : utiliser moins de bits pour les lettres qui apparaissent souvent.

1. Prétraitement : On construit un arbre binaire T qu'on appelle **arbre de Huffman** dont les feuilles sont les lettres.
2. Compression : On code chaque lettre c par une suite de 0 et 1 correspondant au chemin (0 : gauche, 1 : droite) de la racine à la feuille contenant c dans T.

Compression de Huffman

Idée du codage de Huffman : utiliser moins de bits pour les lettres qui apparaissent souvent.

1. Prétraitement : On construit un arbre binaire T qu'on appelle **arbre de Huffman** dont les feuilles sont les lettres.
2. Compression : On code chaque lettre c par une suite de 0 et 1 correspondant au chemin (0 : gauche, 1 : droite) de la racine à la feuille contenant c dans T.
3. Décompression : On décode une suite de 0 et 1 en parcourant le chemin correspondant dans T.

Compression de Huffman : fréquence des caractères

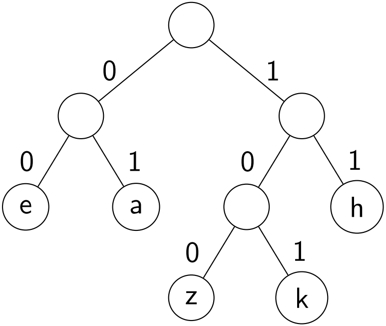
L'algorithme a besoin d'estimer la fréquence d'apparition $f(c)$ de chaque lettre c.
Pour cela, on peut commencer par lire une première fois le texte à coder et compter le nombre d'apparitions de chaque caractère.

Exercice

Écrire une fonction OCaml `get_frequences` comptant les caractères d'un texte et renvoyant un tableau de couples (ou un dictionnaire).

Compression de Huffman : construction de l'arbre

On va construire un arbre dont les étiquettes (lettres) sont aux feuilles et le chemin de la racine à une lettre donne son codage :



Avec cet arbre de Huffman, a est codé par `[0; 1]`, z par `[1; 0; 0]`...

Compression de Huffman : construction de l'arbre

```
type 'a tree = F of 'a | N of 'a tree * 'a tree
```

Compression de Huffman : construction de l'arbre

```
type 'a tree = F of 'a | N of 'a tree * 'a tree
```

Construction de l'arbre de Huffman

q $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c

Compression de Huffman : construction de l'arbre

```
type 'a tree = F of 'a | N of 'a tree * 'a tree
```

Construction de l'arbre de Huffman

q $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c
Tant que q contient ≥ 2 éléments :
  Extraire de q les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`
  Ajouter à q l'arbre `N(t1, t2)` avec la priorité `f1 + f2`

Compression de Huffman : construction de l'arbre

```
type 'a tree = F of 'a | N of 'a tree * 'a tree
```

Construction de l'arbre de Huffman

q $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c
Tant que q contient ≥ 2 éléments :
  Extraire de q les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`
  Ajouter à q l'arbre `N(t1, t2)` avec la priorité `f1 + f2`
L'arbre restant à la fin est l'arbre de Huffman

Compression de Huffman : construction de l'arbre

Construction de l'arbre de Huffman

`q` $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c
Tant que `q` contient ≥ 2 éléments :
    Extraire de `q` les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`
    Ajouter à `q` l'arbre `N(t1, t2)` avec la priorité `f1 + f2`
L'arbre restant à la fin est l'arbre de Huffman

Exemple

Quel arbre de Huffman obtient-on avec les lettres suivantes ?

Lettre	a	b	c	d	e
Fréquence	20	15	7	14	44

Compression de Huffman : construction de l'arbre

Construction de l'arbre de Huffman

`q` $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c
Tant que `q` contient ≥ 2 éléments :
    Extraire de `q` les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`
    Ajouter à `q` l'arbre `N(t1, t2)` avec la priorité `f1 + f2`
L'arbre restant à la fin est l'arbre de Huffman

Question

Quelle est la complexité, en fonction du nombre n de lettres ?

Compression de Huffman : construction de l'arbre

Construction de l'arbre de Huffman

`q` $\longleftarrow$ file de priorité contenant `F(c)` avec la priorité $f(c)$, pour chaque lettre c

Tant que `q` contient ≥ 2 éléments :

    Extraire de `q` les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`

    Ajouter à `q` l'arbre `N(t1, t2)` avec la priorité `f1 + f2`

L'arbre restant à la fin est l'arbre de Huffman

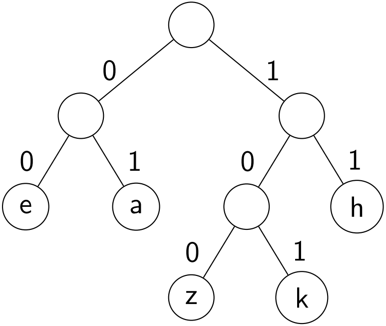
Question

Écrire une fonction

`make_huffman_tree : (int*'a) array -> 'a tree` construisant l'arbre de Huffman associé à un tableau de couples (fréquence, lettre).

Compression de Huffman : table de codage

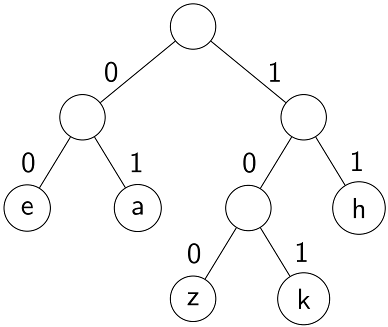
Une fois l'arbre construit, on construit un tableau (ou dictionnaire) associant à chaque lettre une liste de bits.



Avec cet arbre de Huffman, a est codé par `[0; 1]`, z par `[1; 0; 0]`...

Compression de Huffman : table de codage

Une fois l'arbre construit, on construit un tableau (ou dictionnaire) associant à chaque lettre une liste de bits.



Question

Écrire une fonction `make_table` prenant un arbre de Huffman en argument et renvoyant un dictionnaire (construit avec `Map` ou `Hashtbl`) donnant le codage de chaque lettre.

Compression de Huffman : codage

Question

Écrire une fonction `compress_huffman : string -> int list` qui :

1. Calcule les fréquences de chaque lettre : `get_frequences`
2. Calcule l'arbre de Huffman : `make_huffman_tree`
3. Déduire la table de codage : `make_table`
4. Concatène le codage de chaque lettre de la `string`

Compression de Huffman : codage

Théorème

Le codage de Huffman est un codage préfixe, ce qui signifie qu'aucun code d'une lettre n'est préfixe d'un autre code.

Preuve :

Compression de Huffman : codage

Théorème

Le codage de Huffman est un codage préfixe, ce qui signifie qu'aucun code d'une lettre n'est préfixe d'un autre code.

Preuve : On ne peut pas étendre un chemin de la racine en une feuille.

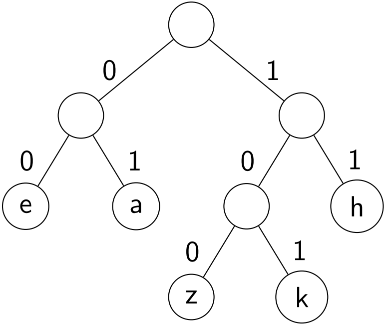
Théorème

Le codage de Huffman est optimal dans le sens où il minimise la longueur moyenne de codage.

Preuve : Voir https://marcdefalco.github.io.

Compression de Huffman : décodage

Pour décoder une suite de bits, il suffit de parcourir l'arbre suivant le bit lu ($0 =$ gauche, $1 =$ droite) et dès qu'on trouve une lettre (une feuille), on l'affiche et on revient à la racine pour le prochain bit.



Exercice

Écrire une fonction
`decode_huffman : char tree -> int list -> string`.

Compression de Huffman : décodage

Pour décoder un ficher compressé par Huffman, il faut stocker l'arbre de Huffman. Pour cela, on peut le **sérialiser** (le convertir en chaîne/liste de caractères).

Compression de Huffman : décodage

Pour décoder un ficher compressé par Huffman, il faut stocker l'arbre de Huffman. Pour cela, on peut le **sérialiser** (le convertir en chaîne/liste de caractères).

Une façon possible est de construire son parcours préfixe en ajoutant un caractère spécial pour reconstruire les feuilles (*) et les noeuds (#) :

```
let rec serialize_tree = function
  | F c -> ['*'; c]
  | N (g, d) -> '#'::(serialize_tree g)@serialize_tree d
```

Compression de Huffman : décodage

Pour décoder un ficher compressé par Huffman, il faut stocker l'arbre de Huffman. Pour cela, on peut le **sérialiser** (le convertir en chaîne/liste de caractères).
Une façon possible est de construire son parcours préfixe en ajoutant un caractère spécial pour reconstruire les feuilles ($*$) et les noeuds (#) :

```
let rec serialize_tree = function
  | F c -> ['*'; c]
  | N (g, d) -> '#'::(serialize_tree g)@serialize_tree d
```

Exercice

Écrire une fonction pour désérialiser un arbre de Huffman.

Lempel-Ziv-Welch (LZW)

Problème : le codage de Huffman demande de calculer la fréquence des lettres du texte entier pour être optimal. Parfois, on a besoin de compresser du texte « en ligne », c'est-à-dire sans attendre d'avoir reçu la totalité.

Lempel-Ziv-Welch (LZW)

Problème : le codage de Huffman demande de calculer la fréquence des lettres du texte entier pour être optimal. Parfois, on a besoin de compresser du texte « en ligne », c'est-à-dire sans attendre d'avoir reçu la totalité.
LZW détermine un codage (dans un dictionnaire d) au fur et à mesure de la lecture du texte. On va coder certains motifs (groupement de lettres consécutives).
Il fonctionne bien quand il y a des motifs qui se répètent.

Lempel-Ziv-Welch (LZW)

LZW détermine un codage (dans un dictionnaire d) pendant la lecture du texte.

Algorithme LZW

Initialement, les n lettres sont codées par un entier (par ex. avec le code ASCII).
Tant qu'il reste du texte s à coder :

- Retirer le plus long préfixe w de s qui soit dans d
- Afficher le codage de w
- $w' \longleftarrow w$ concaténé avec la prochaine lettre de s
- Ajouter un nouveau codage pour w' dans d

Lempel-Ziv-Welch (LZW)

LZW détermine un codage (dans un dictionnaire d) pendant la lecture du texte.

Algorithme LZW

Initialement, les n lettres sont codées par un entier (par ex. avec le code ASCII).
Tant qu'il reste du texte s à coder :
    Retirer le plus long préfixe w de s qui soit dans d
    Afficher le codage de w
    $w' \longleftarrow w$ concaténé avec la prochaine lettre de s
    Ajouter un nouveau codage pour w' dans d

Remarques :

1. Pour décoder, on a besoin de stocker le dictionnaire réciproque de d
2. On peut imposer une longueur maximum d'un motif codé

Lempel-Ziv-Welch (LZW)

Algorithme LZW

Initialement, les n lettres sont codées par un entier (par ex. avec le code ASCII).

Tant qu'il reste du texte s à coder :

  Retirer le plus long préfixe w de s qui soit dans d

  Afficher le codage de w

  $w' \longleftarrow w$ concaténé avec la prochaine lettre de s

  Ajouter un nouveau codage pour w' dans d

Exercice

Appliquer cet algorithme sur $barbapapaba$.

Exercice

Écrire le codage puis le décodage par LZW, en OCaml.