TP9 - Algorithme LZW

Les algorithmes de Huffman et de Lempel–Ziv–Welch (LZW) effectuent des compressions de textes sans perte d'information. Leur principe repose sur l'utilisation de *régularités* dans les textes compressés. L'algorithme de Huffman s'appuie sur la *fréquence des caractères* présents dans le texte ; l'algorithme LZW s'appuie sur la *fréquence de motifs* identifiés dans le texte au fur et à mesure de sa découverte. Ce TP propose une mise en œuvre de ce dernier algorithme.

Principe

Le principe de l'algorithme de compression LZW consiste à rechercher dans le texte à compresser des répétitions de sous-chaînes identiques et à leur donner une forme compacte dans le texte compressé. Il pourrait sembler naturel de commencer par lire intégralement le texte à compresser, à la recherche des fragments qui se répètent. Cependant, l'algorithme LZW procède en une seule passe, en maintenant, au fur et à mesure de la compression, l'ensemble des motifs qu'il a déjà rencontrés. Cette façon de procéder est particulièrement adaptée à la compression d'un document qu'on n'aurait pas le loisir de lire entièrement avant de le compresser, par exemple parce qu'il est trop gros.

Illustrons l'algorithme LZW sur un exemple minimal. Le texte à compresser est la chaîne `"ENTENDENT"`, sur l'alphabet {`E`, `N`, `T`, `D`}. L'algorithme LZW utilise un *dictionnaire*, qui associe à des sous-chaînes du texte à compresser des *codes* qui les représentent. Un code est ici un entier. Le texte compressé sera une suite de codes. Pour démarrer, on associe un code à chaque caractère de notre alphabet, par exemple comme ceci :

dictionnaire	entrée	résultat
`E` → 0; `N` → 1; `T` → 2; `D` → 3	`ENTENDENT`	

On démarre alors la lecture du texte, en considérant le plus grand préfixe du texte qui soit une clé dans le dictionnaire. Ici, ce préfixe se réduit à une lettre, `"E"`. On émet donc le code 0. On regarde alors le prochain caractère de l'entrée, ici `'N'`, et on ajoute au dictionnaire la chaîne `"EN"`, c'est-à-dire la concaténation du préfixe qui a été lu et du caractère qui le suit, avec un nouveau code.

dictionnaire	entrée	résultat
`E` → 0; `N` → 1; `T` → 2; `D` → 3; `EN` → 4	`NTENDENT`	0

L'idée est ici qu'on retombera peut-être sur la chaîne `"EN"` et qu'elle aura alors un code dans le dictionnaire. Ainsi, on représentera plus de caractères avec un seul code. Il n'y a plus qu'à itérer ce processus. Il se passe exactement la même chose à l'étape suivante : le préfixe contenu dans le dictionnaire est réduit à `"N"`, on émet donc le code 1, puis on ajoute la chaîne `"NT"` au dictionnaire.

`E` → 0; `N` → 1; `T` → 2; `D` → 3; `EN` → 4; `NT` → 5 | `TENDENT` | 0 1

C'est toujours la même chose à l'itération suivante, avec le préfixe `"T"` et le caractère suivant `'E'` :

`E` → 0; `N` → 1; `T` → 2; `D` → 3; `EN` → 4; `NT` → 5; `TE` → 6 | `ENDENT` | 0 1 2

Mais les choses deviennent ensuite plus intéressantes à la quatrième itération. En effet, on trouve alors un préfixe de *deux* caractères de l'entrée dans notre dictionnaire, à savoir `"EN"`. On émet alors le code correspondant 4 et on ajoute un nouveau code pour la chaîne `"END"`.

… ; `EN` → 4; `NT` → 5; `TE` → 6; `END` → 7 | `DENT` | 0 1 2 4

On comprend que, de cette façon, des motifs de plus en plus longs vont être ajoutés au dictionnaire et permettre ainsi une efficacité de plus en plus grande des codes. Si plus tard la chaîne `"END"` apparaît de nouveau en position de préfixe, elle sera directement représentée par 7. On poursuit le processus avec l'émission du code 3 pour le préfixe `"D"`,

… ; `EN` → 4; `NT` → 5; `TE` → 6; `END` → 7; `DE` → 8 | `ENT` | 0 1 2 4 3

puis de nouveau du code 4 pour le préfixe `"EN"`,

… ; `EN` → 4; `NT` → 5; `TE` → 6; `END` → 7; `DE` → 8; `ENT` → 9 | `T` | 0 1 2 4 3 4

et enfin du code 2 pour le préfixe `"T"` :

… ; `EN` → 4; `NT` → 5; `TE` → 6; `END` → 7; `DE` → 8; `ENT` → 9 | | 0 1 2 4 3 4 2

Comme on le constate, certains codes n'ont jamais servi, à savoir ici les codes 5 à 9. Mais il n'était pas possible d'en préjuger. Sur un texte plus long, on aurait idéalement trouvé plus de motifs répétés et alors réutilisé plus de codes.

Décompression

Détaillons maintenant le processus de décompression. On a une entrée une séquence de codes et il faut, pour chacun, retrouver la chaîne qui lui est associée. Une solution simple consisterait à inclure le dictionnaire obtenu à la fin de la compression au début du texte compressé. (Plus précisément, c'est le dictionnaire inverse dont on a besoin.) Mais c'est inutile, car *on peut reconstruire le dictionnaire au fur et à mesure de la décompression.* Illustrons-le sur l'exemple précédent. On démarre la décompression avec le même dictionnaire que celui avec lequel on a démarré la compression, c'est-à-dire notre alphabet associé à des codes $0, 1, \ldots$ de manière déterministe.

dictionnaire	entrée	résultat
$0 \to$ `E`; $1 \to$ `N`; $2 \to$ `T`; $3 \to$ `D`	0 1 2 4 3 4 2	

Le premier code étant 0, on émet la chaîne `"E"`.

$0 \to$ `E`; $1 \to$ `N`; $2 \to$ `T`; $3 \to$ `D` | 1 2 4 3 4 2 | `E`

Le deuxième code étant 1, on émet la chaîne `"N"`. Comme l'émission précédente était `"E"`, et que le caractère qui suit est `'N'`, on en déduit que le nouveau code, à savoir 4, est donc associé à `"EN"`, ce que l'on ajoute au dictionnaire.

$0 \to$ `E`; $1 \to$ `N`; $2 \to$ `T`; $3 \to$ `D`; $4 \to$ `EN` | 2 4 3 4 2 | `E N`

À l'étape suivante, on émet `"T"` (code 2) et on ajoute `"NT"` au dictionnaire.

$0 \to$ `E`; $1 \to$ `N`; $2 \to$ `T`; $3 \to$ `D`; $4 \to$ `EN`; $5 \to$ `NT` | 4 3 4 2 | `EN T`

Comme on le voit, la reconstruction du dictionnaire est mécanique : à chaque itération, on ajoute un nouveau code et il est associé à la chaîne décompressée à l'étape précédente suivie du premier caractère de la chaîne décompressée à l'étape courante.

$0 \to$ `E`; $1 \to$ `N`; $2 \to$ `T`; $3 \to$ `D`; $4 \to$ `EN`; $5 \to$ `NT`; $6 \to$ `TE`	3 4 2	`ENT EN`
$\ldots$; $7 \to$ `END`	4 2	`ENTEN D`
$\ldots$; $8 \to$ `DE`	2	`ENTEND EN`
$\ldots$; $9 \to$ `ENT`		`ENTENDEN T`

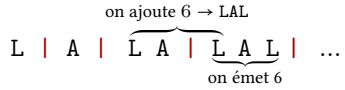
De cette façon, il n'est pas nécessaire d'inclure le dictionnaire dans le format compressé, ce qui constitue un gain de place significatif.
Il y a cependant une subtilité, que l'exemple ci-dessus échoue à illustrer. Il est possible de rencontrer *un code qui n'est encore dans notre dictionnaire !* Illustrons-le avec la compression du texte `"LALALALALERE"`. Elle aboutit au résultat suivant :

dictionnaire	entrée	résultat
`L` $\to 0$; `A` $\to 1$; `E` $\to 2$; `R` $\to 3$	`LALALALALERE`	
`LA` $\to 4$; `AL` $\to 5$; `LAL` $\to 6$; `LALA` $\to 7$	$\ldots$	$\ldots$
`ALE` $\to 8$; `ER` $\to 9$; `RE` $\to 10$	$\ldots$	0 1 4 6 5 2 3 2

(On invite *vraiment* le lecteur à dérouler les étapes de cette compression.) Après trois étapes de décompression, on se retrouve dans la situation suivante,

dictionnaire	entrée	résultat
$0 \to$ `L`; $1 \to$ `A`; $2 \to$ `E`; $3 \to$ `R`; $4 \to$ `LA`; $5 \to$ `AL`	6 5 2 3 2	`LALA`

c'est-à-dire qu'on a décompressé le code 0 en `"L"`, le code 1 en `"A"` puis le code 4 en `"LA"`. Le code suivant est 6 mais il n'est pas encore dans notre dictionnaire. On sait qu'il correspond à la chaîne précédemment décompressée, c'est-à-dire `"LA"`, suivi d'un certain caractère. Mais lequel ? Pour le déterminer, il faut bien visualiser ce qui nous a amenés à cette situation :



Puisque le code qui est utilisé, à savoir 6, est le dernier qui a été construit (il n'est pas encore dans notre dictionnaire), c'est qu'il a été construit avec la dernière chaîne émise, ici `"LA"`, et le premier caractère de la chaîne suivante, *qui est justement la chaîne associée à 6.* Dès lors, le caractère que l'on cherche est forcément *le premier* de la chaîne précédemment émise, ici `'L'`. On est donc toujours en mesure de le déterminer.

Considérations algorithmiques

Pour transformer l'algorithme ci-dessus en un programme effectif, il faut faire plusieurs choix, à commencer par celui d'un alphabet. Une solution consiste à choisir les 256 octets comme autant de caractères et à leur associer les 256 premiers codes. Une autre solution consiste à choisir l'alphabet $\{0, 1\}$ et à lire les bits de l'entrée plutôt que ses octets.
Il faut également décider d'une représentation pour la séquence de codes qui forme le texte compressé. Une solution consiste à choisir une taille fixe pour l'écriture d'un code, par exemple sur 12 bits. Ce choix nous limite à $4\,096$ codes différents. Pour un texte à compresser un tant soit peu long, on parviendra rapidement à cette limite. Là encore, il y a plusieurs options. On peut tout simplement arrêter de remplir le dictionnaire une fois qu'il est plein. Mais on peut également régulièrement *oublier* tous les codes pour repartir de nouveau sur le dictionnaire initial.
Plutôt que d'écrire chaque code avec une taille fixe, on peut également opter pour une *taille variable*. On démarre avec une taille adaptée au dictionnaire initial (8 bits par code pour un alphabet de 256 octets et 1 bit par code pour l'alphabet $\{0, 1\}$) et on incrémente ensuite la taille au fur et à mesure que le dictionnaire grossit. L'idée de tout oublier pour repartir du dictionnaire initial, et donc de la taille initiale, peut également s'appliquer ici.

Implémentation

Le langage de programmation adopté pour implémenter deux fonctions `compress` et `decompress` de compression et de décompression d'un texte est libre : C ou OCaml.

Un fichier texte est mis à votre disposition pour tester vos fonctions.

Un nombre d'octets w variable peut être adopté pour comparer les taux de compression obtenus.

L'initiative est laissé quant à la façon de présenter vos résultats.