Informatique *TP*
Algorithmique des textes 2 Ipesup

1 Huffman

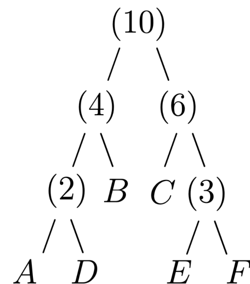
1.1 Exercice

On considère le texte `"ABRACADABRA"`.

1. Donner le nombre d'occurrences de chaque caractère.
2. Construire l'arbre de Huffman correspondant (en cas d'égalité de fréquence, on traitera les arbres par ordre alphabétique du caractère minimal qu'ils contiennent).
3. En déduire le code binaire de chaque caractère. Donner le texte compressé.
4. Calculer le taux de compression $8N/C$ et l'économie
$$E = 1 - \frac{C}{8N}.$$
5. Montrer qu'aucun code préfixe ne peut faire mieux sur ce texte.

1.2 Exercice

On dispose de l'arbre de Huffman suivant, où les feuilles portent des caractères et les nœuds internes leurs poids :



avec les codes :

$$A \mapsto 000, \quad D \mapsto 001, \quad B \mapsto 01, \quad C \mapsto 10, \quad E \mapsto 110, \quad F \mapsto 111.$$

1. Décoder la séquence binaire
$$01\ 000\ 110\ 01\ 001\ 10\ 000\ 111.$$
2. La séquence
$$10110$$
est-elle décodable de manière unique ? Justifier en s'appuyant sur la propriété de code préfixe.
3. Donner un exemple de séquence de longueur 6 qui ne correspond à aucun message valide avec cet arbre, et expliquer pourquoi.

1.3 Exercice

On compresse un texte sur l'alphabet $\{a, b, c, d\}$ avec les fréquences

$$f_a = 0{,}5, \qquad f_b = 0{,}25, \qquad f_c = 0{,}15, \qquad f_d = 0{,}10.$$

1. Construire l'arbre de Huffman et calculer la longueur moyenne
$$S = \sum_i f_i d_i$$
du code.
2. Comparer S au code de longueur fixe sur 2 bits par caractère. Conclure.
3. On remplace f_d par 0,25 et on ajuste $f_a = 0{,}25$ pour conserver
$$\sum_i f_i = 1.$$
Reconstruire l'arbre. La structure a-t-elle changé ? Commenter l'influence de l'équi-répartition sur l'efficacité de Huffman.

1.4 Exercice

On considère le texte `"MISSISSIPPI"`.

1. Donner le nombre d'occurrences de chaque caractère.
2. Construire l'arbre de Huffman, en cas d'égalité, on privilégie l'ordre alphabétique du caractère minimal contenu dans l'arbre. Donner le code binaire de chaque caractère et le texte compressé.
3. Calculer la longueur moyenne
$$S = \sum_i f_i d_i$$
et l'économie E.
4. L'entropie de Shannon de cette distribution est
$$H = -\sum_i f_i \log_2 f_i.$$
Calculer H et comparer à S. Que nous dit cette comparaison sur l'optimalité de Huffman ?
5. [**Code**] La fonction `build_dict` du cours construit les codes par concaténations successives de chaînes, ce qui est coûteux $O(M^2)$ au total. Réécrire en OCaml une version
```
build_dict_eff : tree -> (char, string) Hashtbl.t
```
de coût $O(M)$ en utilisant un accumulateur de type `Buffer.t` pour éviter les concaténations.

1.5 Exercice

On dispose de l'arbre de Huffman suivant avec les codes :

$$A \mapsto 000, \quad D \mapsto 001, \quad B \mapsto 01, \quad C \mapsto 10, \quad E \mapsto 110, \quad F \mapsto 111.$$

1. Décoder la séquence
$$0100011001111 0000111.$$
2. On reçoit la séquence
$$0100011.$$
mais la transmission a été corrompue : un bit a été inversé, sans qu'on ne sache lequel. Lister tous les décodages possibles en testant chaque inversion.
3. Un **code de Huffman canonique** réordonne les codes de sorte que, à longueur égale, ils soient consécutifs et croissants. Transformer les codes ci-dessus en code canonique, avec les longueurs :
$$A : 3, \quad D : 3, \quad B : 2, \quad C : 2, \quad E : 3, \quad F : 3.$$
en appliquant l'algorithme de canonisation. Quel avantage présente-t-il pour stocker l'arbre dans le fichier compressé ?
4. [**Code**] Écrire en OCaml une fonction
```
decode_step : string -> int -> tree -> char * int
```
qui, étant donné le texte compressé s, une position i et l'arbre t, renvoie le prochain caractère décodé et la position suivante, sans utiliser de récursion, avec une boucle `while` et une référence sur le nœud courant.

1.6 Exercice

On compresse un flux en ligne, caractère par caractère, avec l'algorithme de Huffman **adaptatif** : l'arbre est reconstruit après chaque caractère lu, à partir des fréquences mises à jour.

On traite les caractères de `"AABBC"` un par un.

1. Donner l'arbre de Huffman après avoir lu `"A"`, puis `"AA"`, puis `"AAB"`, puis `"AABB"`, puis `"AABBC"`. Pour chaque étape, donner le code utilisé pour le caractère courant.
2. Comparer le nombre total de bits émis avec la compression en une passe sur `"AABBC"`. Conclure sur le surcoût de l'approche adaptative dans ce cas.
3. Quel est l'avantage de l'approche adaptative pour un flux dont on ne connaît pas à l'avance les fréquences, par exemple des données réseau en temps réel ?
4. [**Code**] Écrire en OCaml une fonction
```
update_and_encode : int array -> char -> string
```
qui prend en argument le tableau des occurrences `occ` de taille 256, initialement nul, un caractère `c`, et qui :
   — incrémente `occ.(Char.code c)` ;
   — reconstruit l'arbre de Huffman à partir de `occ` avec `compute_tree` ;
   — renvoie le code binaire de `c` dans ce nouvel arbre.

2 LZW

2.1 Exercice

On considère l'alphabet $\{A, B, C\}$ et le texte `"ABABCABABC"`. On initialise le dictionnaire avec

$$A \mapsto 0, \qquad B \mapsto 1, \qquad C \mapsto 2.$$

1. Dérouler l'algorithme LZW et donner, à chaque étape : le préfixe lu, le code émis, et l'entrée ajoutée au dictionnaire.
2. Donner la séquence de codes produite.
3. Calculer le nombre de bits utilisés si chaque code est écrit sur 3 bits, en taille fixe. Comparer au texte brut sur 8 bits par caractère.
4. À partir de quelle étape la taille de code à 3 bits deviendrait-elle insuffisante si on continuait à compresser un texte plus long ? Que faudrait-il faire ?

2.2 Exercice

On vous donne la séquence de codes LZW suivante :

$$0 \quad 1 \quad 4 \quad 6 \quad 5 \quad 2,$$

avec le dictionnaire initial

$$0 \mapsto L, \qquad 1 \mapsto A, \qquad 2 \mapsto E.$$

1. Dérouler la décompression en reconstruisant le dictionnaire au fil du décodage.
2. À quelle étape rencontre-t-on le cas où le code à décoder n'est pas encore dans le dictionnaire ? Expliquer précisément comment le résoudre.
3. Retrouver le texte original décompressé.
4. Vérifier la cohérence en recompressant manuellement ce texte avec LZW depuis le dictionnaire initial.

2.3 Exercice

1. On souhaite compresser un texte formé de N répétitions du caractère unique `'a'`. Analyser le comportement de l'algorithme de Huffman sur ce texte. Peut-il produire une compression ? Justifier à l'aide du théorème d'optimalité.
2. Analyser le comportement de LZW sur ce même texte. Donner la séquence de codes produite pour $N = 8$ et commenter le taux de compression obtenu.
3. On compresse maintenant un texte aléatoire où chaque caractère parmi 256 est équiprobable. Quel algorithme est le plus adapté, et pourquoi ?
4. Le format ZIP utilise DEFLATE, qui combine Huffman et une variante de LZW. Proposer intuitivement pourquoi cette combinaison peut surpasser chacun des deux algorithmes pris séparément.

2.4 Exercice

On considère l'alphabet $\{A, B, C\}$, le dictionnaire initial

$$A \mapsto 0, \qquad B \mapsto 1, \qquad C \mapsto 2,$$

et le texte `"ABABABCABAB"`.

1. Dérouler la compression LZW pas à pas. Donner à chaque étape le préfixe lu, le code émis et l'entrée ajoutée au dictionnaire.
2. En utilisant une taille de code variable (comme dans le cours), démarrer à 2 bits. Indiquer précisément à quelle étape la taille passe à 3 bits et calculer le nombre total de bits émis.
3. Supposons maintenant que le dictionnaire est limité à 8 entrées (codes 0 à 7). À partir de quelle étape est-il saturé ? Décrire deux stratégies pour gérer cette saturation et discuter leur impact sur la décompression.
4. **[Code]** Dans la version OCaml du cours, la fonction

   `encode`

   utilise un trie (`Trie.prefix_sub`) pour trouver le plus long préfixe. Réécrire la partie centrale de

   `encode`

   en remplaçant le trie par une table de hachage

   `Hashtbl.t`

   de type

   `(string, int) Hashtbl.t`,

   en cherchant le plus long préfixe par extensions successives. Quelle est la complexité de cette nouvelle approche par rapport à celle avec le trie ?

2.5 Exercice

On donne la séquence de codes LZW

$$0 \quad 1 \quad 4 \quad 6 \quad 5 \quad 2 \quad 7,$$

avec le dictionnaire initial

$$0 \mapsto L, \qquad 1 \mapsto A, \qquad 2 \mapsto E.$$

1. Dérouler la décompression en reconstruisant le dictionnaire. Identifier précisément l'étape où un code non encore défini est rencontré et expliquer comment le résoudre en justifiant l'invariant utilisé.
2. Donner le texte décompressé final.
3. Montrer que le cas pathologique (code non encore défini) ne peut se produire que si le code rencontré est exactement le prochain code à définir, et jamais un code encore plus loin. Rédiger une preuve par invariant.

4. **[Code]** La fonction

`decode`

du cours maintient avec une variable

`last`

de type

`string ref`.

Réécrire `decode` en OCaml de façon à lever une exception

`Invalid_argument "LZW: code invalide"`

si un code rencontré est strictement supérieur au prochain code à définir (cas impossible en théorie mais à détecter pour la robustesse), tout en conservant le traitement correct du cas pathologique légitime.

2.6 Exercice

1. On compresse le texte

`"AAAAAAAAAA"`

(10 fois le caractère `'A'`) avec Huffman, puis avec LZW (alphabet $\{A\}$, code initial $A \mapsto 0$, taille variable). Mener les deux compressions jusqu'au bout et calculer l'économie E dans chaque cas. Lequel se comporte le mieux, et pourquoi ?

2. On compresse un texte aléatoire uniforme sur 256 caractères de longueur

$$N = 10^6.$$

Donner un encadrement théorique de S pour Huffman. Montrer que LZW sera en général moins efficace sur ce type de texte.

3. Le format DEFLATE (utilisé dans ZIP et gzip) applique d'abord une étape LZ77 (variante de LZW) qui remplace les répétitions par des références arrières, puis compresse le résultat avec Huffman. Expliquer pourquoi cette composition est plus puissante que chacun des deux algorithmes seuls, en donnant un exemple de texte pour lequel Huffman seul échoue là où DEFLATE réussit.

4. **[Code]** Écrire en OCaml une fonction

`compress_hybrid : string -> string`

qui applique successivement une compression LZW (en réutilisant la fonction

`encode`

du cours, version alphabet $\{0, 1\}$) sur le résultat d'une compression Huffman. La fonction devra appeler

`encode`

de Huffman, récupérer la chaîne de bits, puis appliquer

`encode`

de LZW dessus. Discuter dans quels cas cette double compression est bénéfique et dans quels cas elle est contre-productive.