Informatique *Cours*
Algorithmique des textes - Compression Ipesup

Objectifs

À l'issue de cette section, l'étudiant devra être capable de :
— a

La compression de données consiste à tenter de réduire l'espace occupé par une information. On l'utilise quotidiennement sans le savoir en utilisant des logiciels qui compressent des données pour économiser les ressources. L'exemple typique est celui des formats d'image et de vidéo qui sont le plus souvent compressés. On va illustrer la compression de données avec deux algorithmes, ceci va notamment nous permettre de mettre en pratique les files de priorité et les arbres préfixes.

On suppose que le texte à compresser est une suite de N caractères et que le résultat de la compression est une suite de bits. En pratique, les algorithmes s'appliquent plus généralement à une suite d'octets, voire à une suite de bits. Mais, pour les illustrations au moins, il est plus agréable de supposer que l'on compresse du texte. On fait l'hypothèse que chaque caractère est représenté sur 8 bits. Avec un encodage comme UTF-8, ce n'est pas forcément le cas, mais s'il s'agit de texte dans un alphabet latin, c'est une bonne approximation.

Le résultat de la compression est une suite de C bits. Dans le code que nous allons écrire, on simplifie un peu les choses en construisant une chaîne de `'0'` et `'1'`. En pratique, il faut regrouper ces bits par paquets de huit pour former des octets, avec éventuellement quelques bits de remplissage pour le dernier octet. Le taux de compression est alors le rapport $8N/C$ entre la taille du texte d'origine et celle du texte compressé. On peut le formuler également en termes d'économie d'espace, en considérant

$$E = 1 - \frac{C}{8N}.$$

Ainsi, une économie de $E = 0{,}8$, soit 80%, signifie que le fichier compressé est cinq fois plus petit que le fichier d'origine.

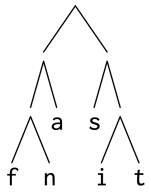
1 Algorithme de Huffman

L'algorithme de Huffman repose sur l'idée suivante : si certains caractères du texte à compresser apparaissent souvent, il est préférable de les représenter par un code court.

Exemple

Dans le texte "satisfaisant", les caractères 'a' et 's' apparaissent souvent. On peut ainsi choisir de représenter le caractère 'a' par 01, le caractère 's' par 10 et les caractères 'f', 'n', 'i' et 't' par des séquences plus longues encore, respectivement 000, 001, 110 et 111. Le texte compressé sera alors 10011111010000011101001001111.

Les séquences pour les différents caractères du texte `"satisfaisant"` n'ont pas été choisies au hasard. Elles ont en effet la propriété qu'aucune n'est un préfixe d'une autre, permettant ainsi un décodage sans ambiguïté. On appelle cela un *code préfixe*. Il se trouve qu'il est très facile de construire un tel code si les caractères considérés forment les feuilles d'un arbre binaire. Prenons par exemple l'arbre suivant :

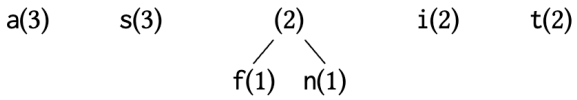


Il suffit alors d'associer à chaque caractère le chemin qui l'atteint depuis la racine, un 0 dénotant une descente vers la gauche et un 1 une descente vers la droite. Par construction, un tel code est un code préfixe.

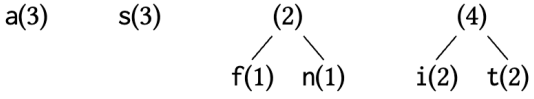
L'algorithme de Huffman permet de construire, étant donné un nombre d'occurrences pour chacun des caractères, un arbre ayant la propriété d'être le meilleur possible pour cette distribution. La fréquence des caractères peut être calculée avec une première passe ou donnée à l'avance s'il s'agit par exemple d'un texte écrit dans un langage pour lequel on connaît la distribution statistique des caractères. Si on reprend l'exemple de la chaîne `"satisfaisant"`, les nombres d'occurrences des caractères sont les suivants :

L'algorithme de Huffman procède alors ainsi.

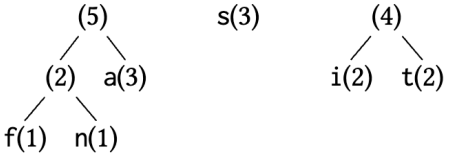
Il sélectionne les deux caractères avec les nombres d'occurrences les plus faibles, à savoir ici les caractères `'f'` et `'n'`, et les réunit en un arbre binaire auquel il donne un nombre d'occurrences égal à la somme des nombres d'occurrences des deux caractères. On a donc la situation suivante :



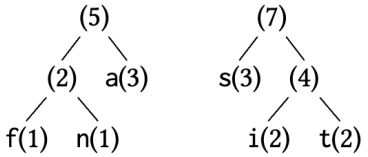
Puis on recommence avec ces nouveaux arbres, c'est-à-dire qu'on en sélectionne deux ayant les occurrences les plus faibles et on les réunit en un nouvel arbre. Ici, on a le choix entre trois arbres de poids 2 et on peut choisir arbitrairement, par exemple comme ceci :



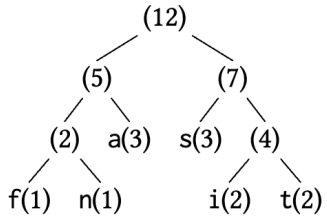
On continue en sélectionnant deux arbres de poids 2 et 3,



puis les deux arbres de poids 3 et 4 :



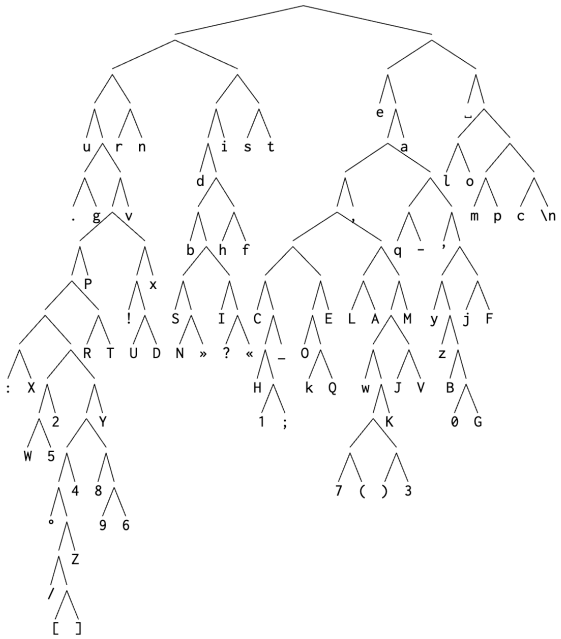
Une dernière étape de ce procédé nous donne au final l'arbre suivant :



C'est l'arbre que nous avions proposé initialement.

Exemple

L'arbre de Huffman sur une œuvre de Jules Verne



La figure illustre l'arbre de Huffman obtenu sur le texte intégral du roman *Le Tour du monde en quatre-vingts jours* de Jules Verne. Il y a 428 775 caractères au total dans ce texte, pour 81 caractères différents (les accents ont été supprimés pour limiter le nombre de caractères). Certains caractères apparaissent souvent dans le texte, comme `'e'` (51 955 occurrences) et `' '` (67 104 occurrences), et l'algorithme a pour effet de les faire apparaître haut dans l'arbre. Ainsi, le caractère `'e'` sera codé par 100 et `' '` par 110. Inversement, certains caractères apparaissent très peu souvent, comme `'Z'` (5 occurrences) ou `'3'` (28 occurrences), et l'algorithme les place dès lors plus profondément dans l'arbre. Ainsi, `'Z'` sera codé par la séquence 0001100001100011 et `'3'` par la séquence 10100011001011.
Au final, le texte compressé occupe 1 919 473 bits, contre 3 430 200 au départ (428 775 × 8 bits par caractères, en supposant un encodage 8 bits type Latin-1), soit une économie de 44%.

Montrons maintenant que l'arbre de Huffman est le meilleur que l'on puisse construire pour un code préfixe. Soit N la taille du texte à compresser et n_i le nombre d'occurrences du caractère c_i. On note $f_i = n_i/N$ la fréquence du caractère c_i.

Théorème. Optimalité de l'arbre de Huffman

L'arbre construit par l'algorithme de Huffman minimise la quantité

$$S = \sum_i f_i \times d_i$$

où d_i est la profondeur du caractère c_i dans l'arbre, c'est-à-dire la longueur du code du caractère c_i dans le texte compressé.

Démonstration. Montrons-le par l'absurde, en supposant qu'il existe un arbre T pour lequel la somme S est strictement plus petite que celle obtenue avec l'algorithme de Huffman. On choisit un tel arbre T qui minimise le nombre n de caractères distincts. On a forcément $n \geq 3$ car pour deux caractères, il n'y a que deux arbres possibles, de même somme, et l'algorithme de Huffman est donc optimal.
Sans perte de généralité, supposons que c_0 et c_1 sont les deux caractères choisis initialement par l'algorithme de Huffman, c'est-à-dire deux caractères avec les fréquences les plus basses. On peut supposer que ces deux caractères sont à la profondeur maximale dans T, car on n'augmente pas la somme S en les échangeant avec des feuilles de profondeur maximale. De même, on peut supposer que ce sont deux feuilles d'un même nœud, car on peut toujours les échanger avec d'autres feuilles de profondeur maximale. Si on remplace alors ce nœud par une feuille de fréquence $f_0 + f_1$, à la fois dans l'arbre T et dans l'arbre de Huffman, la somme S diminue de $f_0 + f_1$ dans les deux arbres (car cette diminution ne dépend pas de la profondeur du nœud). Du coup, on vient de trouver un arbre meilleur que celui donné par l'algorithme de Huffman pour $n - 1$ caractères, ce qui est une contradiction.

On note que la taille C du texte compressé est égale à NS et on en conclut donc que l'algorithme de Huffman minimise la taille du texte compressé (et maximise donc l'économie $E = 1 - S/8$). L'algorithme de Huffman est donc optimal pour un code préfixe.

1.1 Mise en œuvre.

Le programme suivante contient un code OCaml qui compresse une chaîne de caractères avec l'algorithme de Huffman.

Algorithme de Huffman (compression)

```
type tree = Leaf of char | Node of tree * tree

let compute_tree (s: string) : tree =
  let occ = Array.make 256 0 in
  let add c = let i = Char.code c in occ.(i) <- occ.(i) + 1 in
  String.iter add s;
  let q = Pqueue.create () in
  for i = 0 to 255 do
```

```
9     if occ.(i) > 0 then
10      Pqueue.insert q (occ.(i), Leaf (Char.chr i))
11  done;
12  let rec build q =
13    let (n1, t1) = Pqueue.extract_min q in
14    let (n2, t2) = Pqueue.extract_min q in
15    let t = Node (t1, t2) in
16    if Pqueue.is_empty q then t
17    else (Pqueue.insert q (n1 + n2, t); build q) in
18  build q
19
20 (* construit un dictionnaire caractere->code *)
21 let build_dict (t: tree) : (char, string) Hashtbl.t =
22  let d = Hashtbl.create 16 in
23  let rec fill s = function
24    | Leaf c -> Hashtbl.add d c s
25    | Node (l, r) -> fill (s ^ "0") l; fill (s ^ "1") r in
26  fill "" t;
27  d
28
29 let encode (s: string) : tree * string =
30  let t = compute_tree s in
31  let d = build_dict t in
32  let b = Buffer.create 1024 in
33  let encode c = Buffer.add_string b (Hashtbl.find d c) in
34  String.iter encode s;
35  (t, Buffer.contents b)
```

La fonction `compute_tree` construit l'arbre de Huffman. Elle commence par compter les occurrences des caractères (lignes 4–6) puis remplit une file de priorité avec des arbres feuilles, un par caractère apparaissant dans le texte, la priorité étant le nombre d'occurrences (lignes 7–11). La construction de l'arbre est réalisée par la fonction `build` (lignes 12–18).

Pour compresser le texte, il faut associer à chaque caractère le chemin correspondant dans l'arbre. On construit pour cela une table de hachage avec la fonction `build_dict` (lignes 21–27). Les chemins sont construits avec des concaténations de chaînes, ce qui n'est pas très efficace. On pourrait l'améliorer, mais ce n'est pas le propos ici. Enfin, la chaîne s est compressée dans la fonction `encode` (lignes 29–35). On se sert du module `Buffer` de la bibliothèque OCaml pour construire la chaîne de `'0'` et de `'1'`, avant de la renvoyer conjointement avec l'arbre.

Le programme suivante contient le code de décompression.

Algorithme de Huffman (décompression)

```
1 let rec decode1 (s: string) (i: int) (t: tree) : char * int =
2   match t with
3   | Leaf c -> c, i
4   | Node (l, r) -> decode1 s (i+1) (if s.[i]='0' then l else r)
5
6 let decode (t, s : tree * string) : string =
```

```
7   let n = String.length s in
8   let b = Buffer.create 1024 in
9   let rec decode i =
10    if i = n then Buffer.contents b
11    else (let c, i = decode1 s i t in
12          Buffer.add_char b c; decode i) in
13  decode 0
```

La fonction `decode` reçoit en argument l'arbre de Huffman t et la chaîne s à décompresser. Les caractères sont décompressés un par un avec la fonction `decode1`. Elle reçoit en argument une position i dans la chaîne à décompresser. Elle descend dans l'arbre, en suivant les `'0'` et les `'1'`, jusqu'à arriver sur une feuille. Elle renvoie la position qui suit le code qui vient d'être lu. Ainsi, la fonction récursive `decode` (lignes 9–12) avance dans le texte à décompresser avec l'indice i. Là encore, on utilise le module `Buffer` pour construire le résultat. On note que le nombre total de caractères n'a pas besoin d'être indiqué : la décompression se termine lorsqu'on atteint la fin de l'entrée.

Dans notre code, on a conservé séparément l'arbre de Huffman et le texte compressé. En particulier, on peut imaginer un cas d'usage réaliste où l'arbre de Huffman est toujours le même, par exemple parce qu'il est construit à partir de fréquences de caractères fixées une fois pour toutes. On peut alors écrire le texte compressé dans un fichier sans y inclure l'arbre. Mais dans une utilisation de l'algorithme de Huffman qui ne préjuge pas des fréquences des caractères, et les calcule sur le texte à compresser, il faut alors inclure l'arbre au début du texte compressé. Décompresser revient alors à commencer par lire l'arbre, puis à décoder les caractères avec cet arbre.

1.2 Complexité.

Soit N la taille du texte à compresser et M le nombre de caractères distincts dans ce texte. Le décompte des occurrences est en $O(N)$. La construction de l'arbre de Huffman est en $O(M \log M)$, car on fait $O(M)$ retraits et ajouts dans la file de priorité, pour des opérations qui sont toutes en $O(\log M)$. Notre fonction `build_dict` a un coût $O(M^2)$ dans le pire des cas. La compression a un coût directement proportionnel à la taille C du résultat, car on utilise d'une part une table de hachage pour récupérer les codes et d'autre part un tableau redimensionnable (le module `Buffer`) pour stocker le résultat. Dans les deux cas, il s'agit de structures de données avec un coût $O(1)$ amorti par opération. La décompression a également un coût directement proportionnel à C. En effet, pour chaque caractère du texte compressé, on effectue un nombre borné d'opérations, qui soit descendent dans l'arbre, soit ajoutent un caractère au résultat. Là encore, on utilise le module `Buffer` qui garantit un coût $O(1)$ amorti par opération.

2 Algorithme de Lempel–Ziv–Welch

Le second algorithme de compression est connu sous le nom de LZW, les initiales de ses auteurs, Lempel, Ziv et Welch. Le principe de cet algorithme 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. L'algorithme LZW, cependant, procède en une seule passe, en maintenant, au fur et à mesure de la compression, l'ensemble des motifs qu'il a déjà rencontrés. En particulier, c'est adapté 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.

Exemple

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 $\mapsto$ 0 ; N $\mapsto$ 1 ; T $\mapsto$ 2 ; D $\mapsto$ 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 $\mapsto$ 0 ; N $\mapsto$ 1 ; T $\mapsto$ 2 ; D $\mapsto$ 3 ; EN $\mapsto$ 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.

dictionnaire	entrée	résultat
E $\mapsto$ 0 ; N $\mapsto$ 1 ; T $\mapsto$ 2 ; D $\mapsto$ 3 ; EN $\mapsto$ 4 ; NT $\mapsto$ 5	TENDENT	0 1

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

dictionnaire	entrée	résultat
E $\mapsto$ 0 ; N $\mapsto$ 1 ; T $\mapsto$ 2 ; D $\mapsto$ 3 ; EN $\mapsto$ 4 ; NT $\mapsto$ 5 ; TE $\mapsto$ 6	NDENT	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"`.

dictionnaire	entrée	résultat
... ; EN $\mapsto$ 4 ; NT $\mapsto$ 5 ; TE $\mapsto$ 6 ; END $\mapsto$ 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"` :

dictionnaire	entrée	résultat
... ; EN $\mapsto$ 4 ; NT $\mapsto$ 5 ; TE $\mapsto$ 6 ; END $\mapsto$ 7 ; DE $\mapsto$ 8	ENT	0 1 2 4 3

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

dictionnaire	entrée	résultat
... ; NT $\mapsto$ 5 ; TE $\mapsto$ 6 ; END $\mapsto$ 7 ; DE $\mapsto$ 8 ; ENT $\mapsto$ 9	T	0 1 2 4 3 4

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

dictionnaire	entrée	résultat
... ; TE $\mapsto$ 6 ; END $\mapsto$ 7 ; DE $\mapsto$ 8 ; ENT $\mapsto$ 9	$\emptyset$	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 en 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. Mais c'est inutile, car on peut reconstruire le dictionnaire au fur et à mesure de la décompression.

Exemple

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 $\mapsto$ E ; 1 $\mapsto$ N ; 2 $\mapsto$ T ; 3 $\mapsto$ D	0 1 2 4 3 4 2	

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

$0 \mapsto E;\ 1 \mapsto N;\ 2 \mapsto T;\ 3 \mapsto D$	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.

$0 \mapsto E;\ 1 \mapsto N;\ 2 \mapsto T;\ 3 \mapsto D;\ 4 \mapsto EN$	2 4 3 4 2	EN

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

$$0 \mapsto E;\ 1 \mapsto N;\ 2 \mapsto T;\ 3 \mapsto D;\ 4 \mapsto EN;\ 5 \mapsto NT \quad \Big| \quad 4\,3\,4\,2 \quad \Big| \quad ENT$$

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.

$$\ldots;\ 6 \mapsto TE \quad \Big| \quad 3\,4\,2 \quad \Big| \quad ENTEN$$

$$\ldots;\ 7 \mapsto END \quad \Big| \quad 4\,2 \quad \Big| \quad ENTEND$$

$$\ldots;\ 8 \mapsto DE \quad \Big| \quad 2 \quad \Big| \quad ENTENDEN$$

$$\ldots;\ 9 \mapsto ENT \quad \Big| \quad \emptyset \quad \Big| \quad ENTENDENT$$

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 pas encore dans notre dictionnaire !

Exemple

La compression du texte `"LALALALALERE"` aboutit au résultat suivant :

dictionnaire	entrée	résultat
L ↦ 0 ; A ↦ 1 ; E ↦ 2 ; R ↦ 3	LALALALALERE	… 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 ↦ L ; 1 ↦ A ; 2 ↦ E ; 3 ↦ R ; 4 ↦ LA ; 5 ↦ 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 :

$$\text{on ajoute } 6 \mapsto LAL \qquad \text{et on émet } 6$$

Puisque le code qui est utilisé, à savoir 6, est le dernier qui a été construit (il n'est pas encore dans le 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.

2.1

Mise en œuvre.

Considérations algorithmiques. Pour transformer l'algorithme en un programme effectif, il faut faire plusieurs choix.

— Pour l'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.

— Pour la 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 4096 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.

On présente une implémentation avec un alphabet $\{0, 1\}$ et une taille de code variable.

Programmes Les programmes suivantes contiennent un code OCaml pour l'algorithme LZW. La chaîne à compresser ou à décompresser est une chaîne OCaml `s` de type `string` ne contenant que des caractères `'0'` et `'1'`.

```
(* crit `c` sur `n` chiffres en binaire dans le buffer `b` *)
let rec output (b: Buffer.t) (n: int) (c: int) : unit =
  if n > 0 then (
    output b (n - 1) (c / 2);
    Buffer.add_char b (if c mod 2 = 1 then '1' else '0')
  )

let encode (s: string) : string =
  let n = String.length s in
  let codes = Trie.create () in
  Trie.put codes "0" 0;
  Trie.put codes "1" 1;
  let size = ref 1 in
  let next = ref 2 in
  let b = Buffer.create 1024 in
  let i = ref 0 in
  while !i < n do
    let (p, c) = Trie.prefix_sub codes s !i in
    output b !size c;
    if !i + p < n then (
      let w = String.sub s !i (p + 1) in
      Trie.put codes w !next;
```

```
23      if !next = 1 lsl !size then incr size;
24      incr next
25    );
26    i := !i + p
27  done;
28  Buffer.contents b
```

Pour la compression, on utilise un arbre préfixe pour le dictionnaire. Initialement, il contient uniquement les codes pour les caractères `'0'` et `'1'` (lignes 10–12). La variable `next` contient le numéro du prochain code et la variable `size` contient la taille, en nombre de bits, utilisée pour émettre les codes. Le résultat est construit avec le module `Buffer` (ligne 15). On avance dans le texte à compresser avec la variable `i` (ligne 16). On consulte le dictionnaire avec une fonction `Trie.prefix_sub` qui détermine le plus grand préfixe de `s[i..[` qui est une clé dans le dictionnaire. Le code est émis avec la fonction auxiliaire `output` (lignes 2–6) qui écrit l'entier `c` sur `size` bits dans le résultat. Si la fin de l'entrée n'est pas atteinte (ligne 20), on étend le dictionnaire avec un nouveau code (lignes 21–22). Et lorsque `next` est une puissance de deux, on incrémente `size` (ligne 23).

```
1  (* lit `size` bits au dbut de `s[i..[` *)
2  let input (size: int) (s: string) (i: int) : int =
3    assert(i + size <= String.length s);
4    int_of_string ("0b" ^ String.sub s i size)
5
6  let decode (s: string) : string =
7    let words = Vector.create 2 "" in
8    Vector.push words "0";
9    Vector.push words "1";
10   let size = ref 1 in
11   let b = Buffer.create 1024 in
12   let i = ref 0 in
13   let last = ref "" in
14   while !i < String.length s do
15     let c = input !size s !i in
16     i := !i + !size;
17     if !last <> "" then (
18       let next = Vector.size words in
19       let w =
20         if c = next then !last
21         else Vector.get words c
22       in
23       Vector.push words (!last ^ String.sub w 0 1)
24     );
25     if Vector.size words = 1 lsl !size then incr size;
26     last := Vector.get words c;
27     Buffer.add_string b !last
28   done;
29   Buffer.contents b
```

Pour la décompression, on se sert d'un tableau redimensionnable pour contenir le dictionnaire inversé. Comme pour la compression, le résultat est construit avec le module `Buffer` (ligne 11), on avance dans la chaîne à décompresser avec une variable `i` (ligne 12) et la variable `size` indique la taille des codes (lignes 10). On lit un entier sur `size` bits à la position `i` avec

la fonction auxiliaire `input` (lignes 2–4). Toute la subtilité est concentrée dans les lignes 17 à 23. On commence par déterminer le nouveau code à ajouter au dictionnaire, sauf s'il s'agit de la toute première itération (test ligne 18). Le nouveau code correspond au mot précédemment émis, contenu dans la variable `last`, auquel on rajoute le premier caractère de ce que l'on s'apprête à émettre. Il y a une difficulté lorsque le code `c` est justement le prochain, dont nous avons discuté plus haut, prise en charge par la ligne 19. Ensuite, on détermine si la taille doit être incrémentée (ligne 22), on met à jour `last` (ligne 23) puis enfin on émet la portion de résultat (ligne 24).

Exemple

On reprend l'exemple du texte *Le Tour du monde en quatre-vingts jours* que nous avions compressé avec l'algorithme de Huffman. Avec LZW on obtient au final un texte compressé de 2 438 793 bits, soit une économie $E = 29\%$, ce qui est moins bon qu'avec l'algorithme de Huffman ($E = 44\%$). Au total, on a produit 150 052 codes, utilisant jusqu'à 18 bits. Beaucoup ne sont jamais réutilisés, mais ils ont pour autant comme effet d'augmenter la taille occupée par chaque code écrit dans le résultat.
Si on adopte plutôt un alphabet de 256 octets, d'une part, et que l'on fixe la taille des codes à W bits, d'autre part, alors on parvient à une économie de 59% en optant pour $W = 16$.

Inversement, certains textes seront mieux compressés avec l'approche par taille variable. Tout va dépendre de la forme du texte à compresser.

Le format ZIP

Le format de compression ZIP bien connu offre le choix entre plusieurs algorithmes, mais le plus répandu est DEFLATE, un algorithme qui combine les algorithmes de Huffman et de Lempel–Ziv–Welch (plus précisément un prédécesseur de l'algorithme LZW).

La compression avec perte

Les algorithmes de compression que nous avons présentés sont dits *sans perte*, c'est-à-dire que la donnée décompressée est identique à la donnée départ. Lorsqu'il s'agit de texte, c'est tout à fait souhaitable. Mais lorsque l'on compresse des données comme du son, de l'image ou de la vidéo, il peut devenir intéressant d'autoriser de la perte dans la compression, pour obtenir un meilleur taux de compression. C'est le cas notamment de formats comme MP3 pour le son ou JPEG pour l'image.