Tache 1

Khalil Mellouk

March 2024

1 Run-Length Encoding (RLE)

1.1 Introduction

Les fichiers de données contiennent souvent le même caractère répété plusieurs fois de suite. Par exemple, les fichiers texte utilisent plusieurs espaces pour séparer les phrases, mettre en retrait des paragraphes, formater des tableaux et des graphiques, etc. Les signaux numérisés peuvent également avoir des séquences de même valeur, indiquant que le signal ne change pas. Par exemple, une image du ciel nocturne contiendrait de longues séquences du ou des personnages représentant le fond noir. De même, la musique numérisée peut comporter une longue série de zéros entre les chansons. Run-Length Encoding est une méthode simple pour compresser ces types de fichiers.

1.2 Méthodologie

La figure 1 illustre Run-Length Encoding pour une séquence de données comportant de fréquentes séries de zéros. Chaque fois qu'un zéro est rencontré dans les données d'entrée, deux valeurs sont écrites dans le fichier de sortie. La première de ces valeurs est un zéro, un indicateur indiquant que la compression de longueur d'exécution commence. La deuxième valeur est le nombre de zéros dans l'exécution. Si la longueur moyenne est supérieure à deux, une compression aura lieu. D'un autre côté, de nombreux zéros simples dans les données peuvent rendre le fichier codé plus volumineux que l'original.

1.3 Variantes de run-length Encoding

De nombreux schémas de run-length ont été développés. Par exemple, les données d'entrée peuvent être traitées comme des octets individuels ou des groupes d'octets représentant quelque chose de plus élaboré, comme des nombres à virgule flottante. Run-Length Encoding peut être utilisé sur un seul des caractères (comme pour le zéro ci-dessus), sur plusieurs caractères ou sur tous les caractères. Un bon exemple de schéma d'exécution généralisé est PackBits, créé pour les utilisateurs Macintosh.

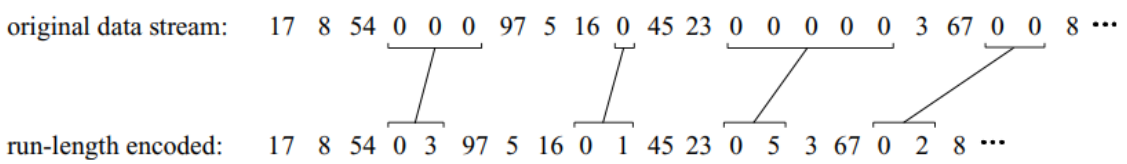

Figure 1: Exemple run-length Encoding. Chaque série de zéros est remplacée par deux caractères dans le fichier compressé : un zéro pour indiquer que la compression est en cours, suivi du nombre de zéros dans la série.

1.3.1 Algorithme de compression PackBits

PackBits est un schéma RLE généralisé développé pour les utilisateurs Macintosh. Chaque octet (huit bits) du fichier d'entrée est remplacé par neuf bits dans le fichier compressé. Le neuvième bit ajouté est interprété comme le signe du nombre. Autrement dit, chaque caractère lu dans le fichier d'entrée est compris entre 0 et 255, tandis que chaque caractère écrit dans le fichier codé est compris entre -255 et 255.

1.3.2 Illustration des PackBits

Pour comprendre comment PackBits est utilisé, considérons le fichier d'entrée : , et 1,2,3,4,2,2,2,2,4 le fichier compressé généré par l'algorithme PackBits : Le 1,2,3,4,2 ,3,4. Le programme de compression transfère simplement chaque numéro du fichier d'entrée vers le fichier compressé, à l'exception de l'exécution : 2,2,2,2. Ceci est représenté dans le fichier compressé par les deux nombres : 2,-3. Le premier chiffre ("2") indique de quel caractère se compose la course. Le deuxième nombre ("-3") indique le nombre de caractères dans l'exécution, trouvé en prenant la valeur absolue et en en ajoutant une. Par exemple, 4,-2 signifie 4,4,4 ; 21,-4 signifie 21,21,21,21,21, etc.

1.3.3 Modification pour le texte ASCII

Un inconvénient avec PackBits est que les neuf bits doivent être reformatés en octets standard de huit bits utilisés dans le stockage et la transmission informatiques. Une modification utile à ce schéma peut être apportée lorsque l'entrée est limitée au texte ASCII. Comme le montre le tableau 27-2, chaque caractère ASCII est généralement stocké sous la forme d'un octet complet (huit bits), mais n'utilise en réalité que sept bits pour identifier le caractère. En d'autres termes, les valeurs 127 à 255 ne sont définies avec aucune signification standardisée et n'ont pas besoin d'être stockées ou transmises. Cela permet au huitième bit d'indiquer si le codage de longueur de plage est en cours.

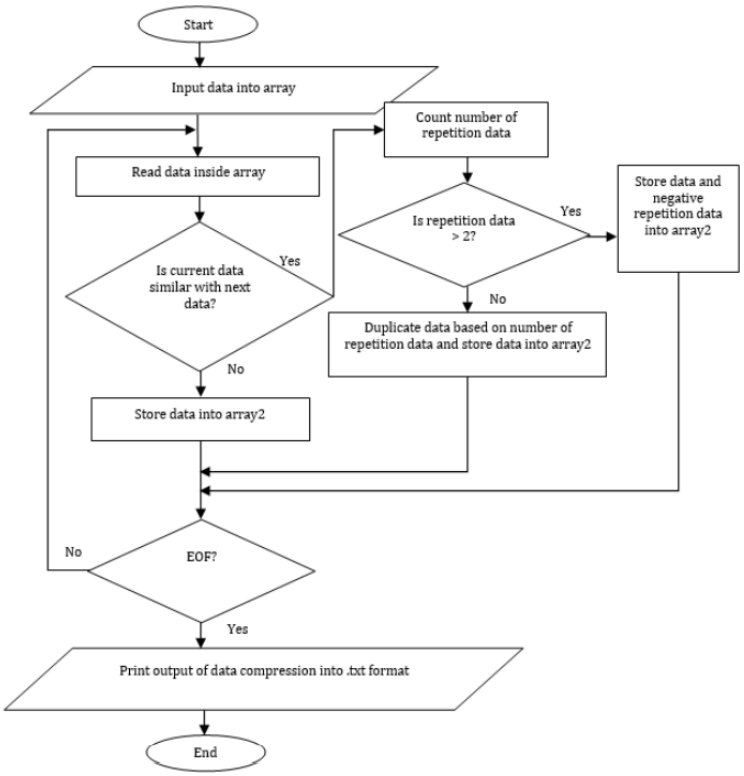

Figure 2: Flowchart of RLE compression algorithm

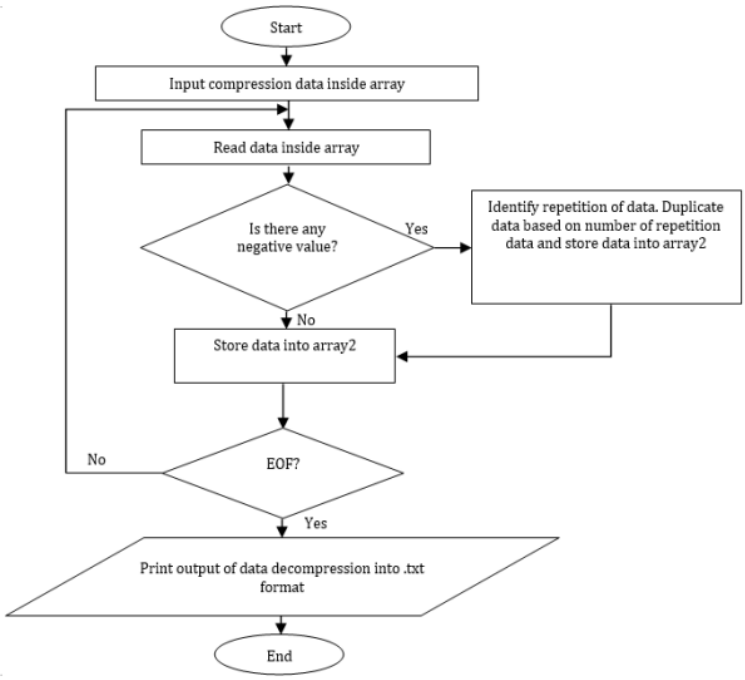

Figure 3: Flowchart of RLE decompression algorithm

2 Huffman

2.1 Introduction

Cette méthode porte le nom de D.A. Huffman, qui a développé cette procédure dans les années 1950. La figure 27-2 montre un histogramme des valeurs d'octets d'un gros fichier ASCII. Plus de 0,96*100 de ce fichier est constitué de seulement 31 caractères : les lettres minuscules, l'espace, la virgule, le point et le retour chariot. Cette observation peut être utilisée pour créer un schéma de compression approprié pour ce fichier.

2.2 Schéma de compression de base

Pour commencer, nous allons attribuer à chacun de ces 31 caractères communs un code binaire de cinq bits : 00000 = "a", 00001 = "b", 00010 = "c", etc. Cela permet de réduire 0,96*100 du fichier en taille par 5/8. Le dernier des codes à cinq bits, 11111, sera un indicateur indiquant que le caractère transmis ne fait pas partie des 31 caractères communs. Les huit bits suivants du fichier indiquent quel est le caractère, selon l'affectation ASCII standard.
Cela se traduit par 0,04*100 des caractères du fichier d'entrée nécessitant 5+8=13 bits. L'idée est d'attribuer moins de bits aux caractères fréquemment utilisés et plus de bits aux caractères rarement utilisés. Dans cet exemple, le nombre moyen de bits requis par caractère original est calculé.

2.3 Principes de codage de Huffman

Le codage Huffman pousse cette idée à l'extrême, où les caractères qui apparaissent le plus souvent, tels que l'espace et le point, peuvent se voir attribuer seulement un ou deux bits. Les caractères rarement utilisés peuvent nécessiter une douzaine de bits ou plus. La situation optimale est atteinte lorsque le nombre de bits utilisés pour chaque caractère est proportionnel au logarithme de la probabilité d'apparition du caractère.

2.4 Codes de longueur variable et emballage

Une fonctionnalité intelligente du codage Huffman est la façon dont les codes de longueur variable peuvent être regroupés. Les codes de longueur variable sont regroupés en groupes de huit bits, la norme pour une utilisation informatique, permettant une séparation correcte lors de la décompression.

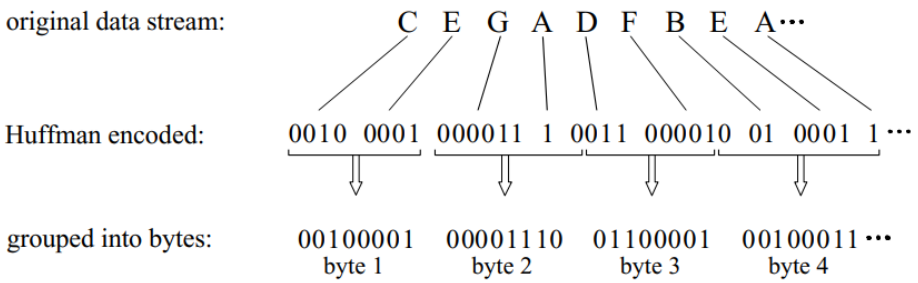

Figure 4: schéma de codage Huffman simplifié, illustrant comment la probabilité d'occurrence de chaque caractère influence la longueur de code qui lui est attribuée.

La table de codage attribue à chacune des sept lettres utilisées dans cet exemple un code binaire de longueur variable, en fonction de sa probabilité d'occurrence. Le flux de données original composé de ces 7 caractères est traduit par cette table en données codées par Huffman. Étant donné que chacun des codes de Huffman a une longueur différente, les données binaires doivent être regroupées en octets standard de 8 bits pour le stockage et la transmission.

Example Encoding Table

letter	probability	Huffman code
A	.154	1
B	.110	01
C	.072	0010
D	.063	0011
E	.059	0001
F	.015	000010
G	.011	000011

Figure 5: Encoding Table

2.5 Gestion du flux de données

Lors de la décompression, tous les groupes de huit bits sont placés bout à bout pour former une longue chaîne série de uns et de zéros. La manière dont les codes sont formés garantit qu'aucune ambiguïté n'existe dans la séparation.

2.6 Codage arithmétique

Une version plus sophistiquée de l'approche de Huffman est appelée codage arithmétique, où les séquences de caractères sont représentées par des codes individuels, en fonction de leur probabilité d'occurrence, ce qui permet une meilleure compression des données.

2.7 Combinaison de méthodes d'encodage

Le codage en longueur suivi d'un codage Huffman ou arithmétique est également une stratégie courante, conduisant à une compression plus efficace.

2.8 Considérations de mise en œuvre

La mise en œuvre du codage Huffman ou arithmétique nécessite un accord sur les codes binaires utilisés pour représenter chaque caractère. Cela peut être géré à l'aide de tables de codage prédéfinies ou d'un codage optimisé pour les données particulières utilisées. Les deux méthodes sont courantes.

3 LZW

3.1 Introduction

La compression LZW porte le nom de ses développeurs, A. Lempel et J. Ziv, avec des modifications ultérieures par Terry A. Welch. Il s'agit de la principale technique de compression de données à usage général en raison de sa simplicité et de sa polyvalence.

3.2 Efficacité de compression du LZW

En règle générale, vous pouvez vous attendre à ce que LZW compresse le texte, le code exécutable et les fichiers de données similaires à environ la moitié de leur taille d'origine. LZW fonctionne également bien lorsqu'il est présenté avec des fichiers de données extrêmement redondants, tels que des nombres tabulés, du code source informatique et des signaux acquis. Des taux de compression de 5:1 sont courants dans ces cas.

3.3 Applications et utilisation de LZW

LZW est à la base de plusieurs utilitaires informatiques personnels qui prétendent doubler la capacité de votre disque dur . La compression LZW est toujours utilisée dans les fichiers image GIF et proposée en option dans TIFF et PostScript.

3.4 Détails techniques de la compression LZW

La compression LZW utilise une table de codes, comme illustré sur la figure 6. Un choix courant consiste à fournir 4 096 entrées dans le tableau. Dans ce cas, les données codées LZW sont entièrement constituées de codes de 12 bits, chacun faisant référence à l'une des entrées de la table de codes.

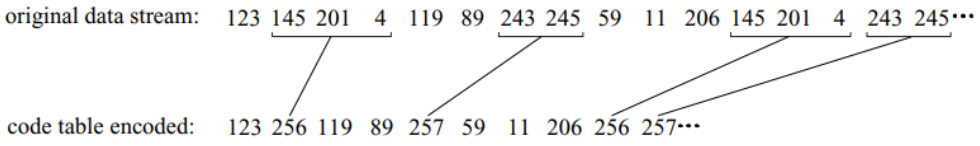

Figure 6: Cogade LZW

C'est la base de la méthode de compression populaire LZW. Le codage s'effectue en identifiant les séquences d'octets du fichier d'origine qui existent dans la table de codes. Le code de 12 bits représentant la séquence est placé dans le fichier compressé à la place de la séquence. Les 256

premières entrées du tableau correspondent aux valeurs d'un seul octet, de 0 à 255, tandis que les entrées restantes correspondent à des séquences d'octets. L'algorithme LZW est un moyen efficace de générer la table de codes en fonction des données particulières compressées. (La table de codes de cette figure est un exemple simplifié, et non réellement généré par l'algorithme LZW).

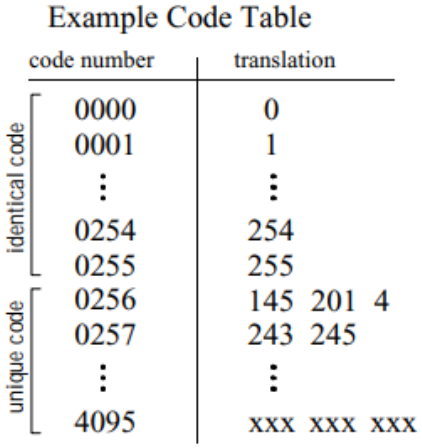

Figure 7: Exemple de compression de table de codes

3.5 Processus d'encodage et de décodage

La décompression est obtenue en prenant chaque code du fichier compressé et en le traduisant via la table de codes pour trouver le ou les caractères qu'il représente. Les codes 0 à 255 dans la table de codes sont toujours attribués pour représenter des octets uniques du fichier d'entrée.

3.6 Défis et considérations

Par exemple, si seuls ces 256 premiers codes étaient utilisés, chaque octet du fichier d'origine serait converti en 12 bits dans le fichier codé LZW, ce qui entraînerait une taille de fichier 0.50*100 plus grande. Lors de la décompression, chaque code de 12 bits serait traduit via la table de codes en octets simples. Bien sûr, cette situation ne serait pas utile.
bien qu'il s'agisse d'une approche simple, deux obstacles majeurs doivent être surmontés: (1) comment déterminer quelles séquences doivent figurer dans la table de codes et (2) comment fournir au programme de décompression le même code. table utilisée par le programme de compression.

3.7 Solution avec l'algorithme LZW

L'algorithme LZW résout de manière exquise ces deux problèmes. Lorsque le programme LZW commence à coder un fichier, la table de codes contient uniquement les 256 premières entrées, le reste de la table étant vide. Cela signifie que les premiers codes entrant dans le fichier compressé sont simplement les octets uniques du fichier d'entrée convertis en 12 bits.

3.8 Ajout de table de codes dynamique

au fur et à mesure que l'encodage se poursuit, l'algorithme LZW identifie les séquences répétées dans les données et les ajoute à la table de codes. La compression démarre la deuxième fois qu'une séquence est rencontrée. Le point clé est qu'une séquence du fichier d'entrée n'est ajoutée à la table de codes que lorsqu'elle a déjà été placée dans le fichier compressé sous forme de caractères individuels (codes 0 à 255).

3.9 Avantages pour la décompression

Ceci est important car cela permet au programme de décompression de reconstruire la table de codes directement à partir des données compressées, sans avoir à transmettre la table de codes séparément.

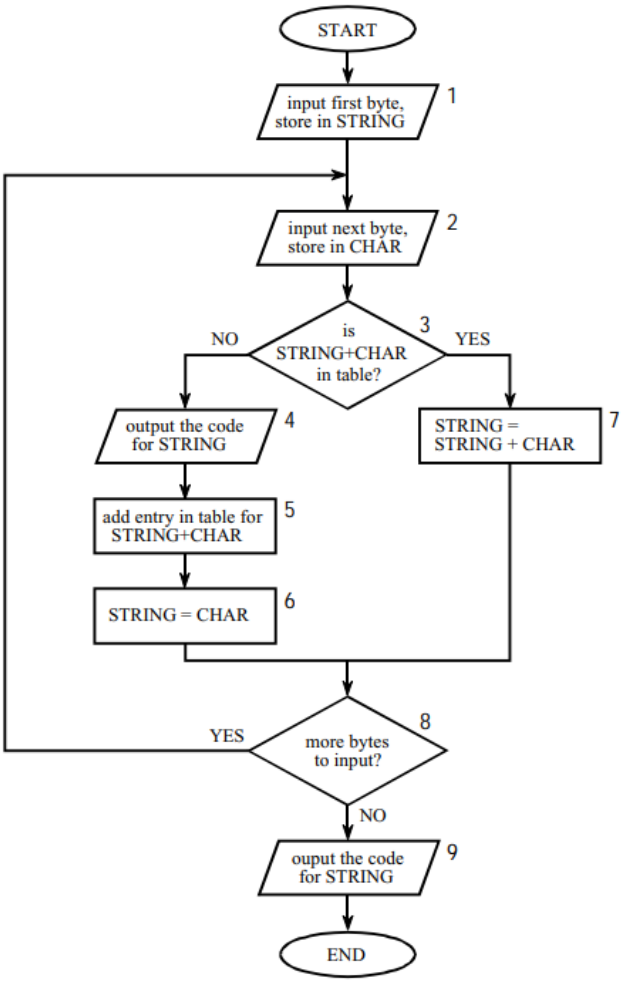

Figure 8: LZW compression flowchart. The variable, CHAR, is a single byte. The variable, STRING, is a variable length sequence of bytes. Data are read from the input file (box 1 2) as single bytes, and written to the compressed file (box 4) as 12 bit codes. Table 27-3 shows an example of this algorithm

Figure 9 provides the step-by-step details for an example input file consisting of 45 bytes, the ASCII text string: the/rain/in/Spain/falls/mainly/on/the/plain. When we say that the LZW algorithm reads the character "a" from the input file, we mean it reads the value: 01100001 (97 expressed in 8 bits), where 97 is "a" in ASCII. When we say it writes the character "a" to the encoded file, we mean it writes: 000001100001 (97 expressed in 12 bits).

	CHAR	*STRING + CHAR*	In Table?	Output	Add to Table	New *STRING*	Comments
1	t	t				t	first character- no action
2	h	th	no	t	256 = th	h	
3	e	he	no	h	257 = he	e	
4	/	e/	no	e	258 = e/	/	
5	r	/r	no	/	259 = /r	r	
6	a	ra	no	r	260 = ra	a	
7	i	ai	no	a	261 = ai	i	
8	n	in	no	i	262 = in	n	
9	/	n/	no	n	263 = n/	/	
10	i	/i	no	/	264 = /i	i	
11	n	in	yes (262)			in	first match found
12	/	in/	no	262	265 = in/	/	
13	S	/S	no	/	266 = /S	S	
14	p	Sp	no	S	267 = Sp	p	
15	a	pa	no	p	268 = pa	a	
16	i	ai	yes (261)			ai	matches *ai*, *ain* not in table yet
17	n	ain	no	261	269 = ain	n	*ain* added to table
18	/	n/	yes (263)			n/	
19	f	n/f	no	263	270 = n/f	f	
20	a	fa	no	f	271 = fa	a	
21	l	al	no	a	272 = al	l	
22	l	ll	no	l	273 = ll	l	
23	s	ls	no	l	274 = ls	s	
24	/	s/	no	s	275 = s/	/	
25	m	/m	no	/	276 = /m	m	
26	a	ma	no	m	277 = ma	a	
27	i	ai	yes (261)			ai	matches *ai*
28	n	ain	yes (269)			ain	matches longer string, *ain*
29	l	ainl	no	269	278 = ainl	l	
30	y	ly	no	l	279 = ly	y	
31	/	y/	no	y	280 = y/	/	
32	o	/o	no	/	281 = /o	o	
33	n	on	no	o	282 = on	n	
34	/	n/	yes (263)			n/	
35	t	n/t	no	263	283 = n/t	t	
36	h	th	yes (256)			th	matches *th*, *the* not in table yet
37	e	the	no	256	284 = the	e	*the* added to table
38	/	e/	yes			e/	
39	p	e/p	no	258	285 = e/p	p	
40	l	pl	no	p	286 = pl	l	
41	a	la	no	l	287 = la	a	
42	i	ai	yes (261)			ai	matches *ai*
43	n	ain	yes (269)			ain	matches longer string *ain*
44	/	ain/	no	269	288 = ain/	/	
45	EOF	/		/			end of file, output *STRING*

Figure 9: Exemple de compression avec l'algorithme LZW. Cela montre la compression de la phrase : "the/rain/in/Spain/falls/mainly/on/the/plain/."

4 L'algorithme de compression LZW

L'algorithme de compression utilise deux variables : CHAR et STRING. La variable CHAR contient un seul caractère, c'est-à-dire une valeur d'un seul octet comprise entre 0 et 255. La variable STRING est une chaîne de longueur variable, c'est-à-dire un groupe d'un ou plusieurs caractères, chaque caractère étant un seul octet.

4.1 Aperçu du processus de compression

Dans la case 1 de la figure 8, le programme commence par prendre le premier octet du fichier d'entrée et le placer dans la variable STRING. L'algorithme boucle ensuite pour chaque octet supplémentaire dans le fichier d'entrée, contrôlé dans l'organigramme par la case 8. Chaque fois qu'un octet est lu dans le fichier d'entrée (case 2), il est stocké dans la variable CHAR. La table de données est ensuite parcourue pour déterminer si la concaténation des deux variables, STRING+CHAR, s'est déjà vue attribuer un code (case 3).

4.2 Gestion des correspondances et des non-correspondances dans la table de codes

Si aucune correspondance dans la table de codes n'est trouvée, trois actions sont entreprises : écrire le code de 12 bits correspondant au contenu de la variable STRING dans le fichier compressé (case 4), créer un nouveau code dans la table pour la concaténation de STRING+CHAR (case 5) et mise à jour de la variable STRING pour prendre la valeur de la variable CHAR (case 6).

4.3 Exemple d'illustration du processus de compression

Un exemple de ces actions est présenté aux lignes 2 à 10 de la figure 8, pour les 10 premiers octets du fichier exemple. Lorsqu'une correspondance dans la table de codes est trouvée (case 3), la concaténation de STRING+CHAR est stockée dans la variable STRING, sans qu'aucune autre action n'ait lieu (case 7).

4.4 Gestion de séquences de correspondance plus longues

Si une séquence correspondante plus longue est trouvée dans le tableau, le programme l'ajoute au tableau, génère le code de la séquence la plus courte qui se trouve dans le tableau et recommence la recherche des séquences commençant par le caractère lu dans le fichier d'entrée.

4.5 Conclusion du processus de compression

Le programme se termine par l'écriture du code correspondant à la valeur actuelle de STRING dans le fichier compressé (comme illustré dans la case 9 de la figure 8 et la ligne 45 de la figure 9).

5 Présentation de l'algorithme de décompression

Un organigramme de l'algorithme de décompression LZW est présenté sur la figure 10. Chaque code est lu à partir du fichier compressé et comparé à la table de codes pour fournir la traduction. Au fur et à mesure que chaque code est traité de cette manière, la table de codes est mise à jour afin qu'elle corresponde continuellement à celle utilisée lors de la compression.

5.1 Gestion des imprévus lors de la décompression

Certaines combinaisons de données font que l'algorithme de décompression reçoit un code qui n'existe pas encore dans sa table de codes. Cette éventualité est gérée dans les cases 4, 5 et 6 de la routine de décompression.

5.2 Gestion efficace de la table de codes

Une gestion efficace de la table de codes est cruciale pour les programmes LZW. Diverses techniques sont utilisées pour optimiser l'utilisation de la mémoire et la vitesse d'exécution du programme, comme l'exploitation de la nature redondante de la table de codes.

5.3 Nature compétitive du champ de compression

LZW et les schémas de compression similaires sont des domaines hautement compétitifs. Les produits commerciaux sont sophistiqués et protégés par des brevets et des secrets commerciaux, ce qui rend difficile l'obtention du même niveau de performance en peu de temps.

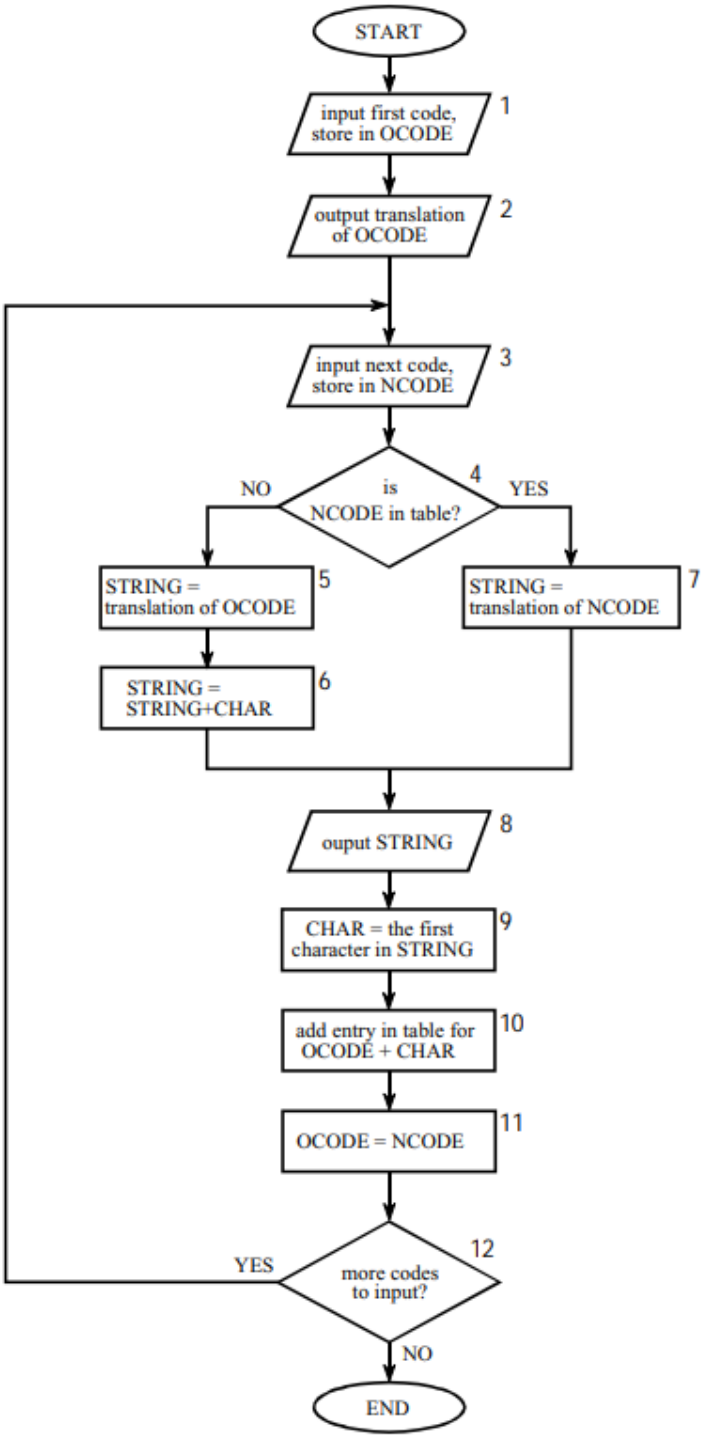

Figure 10: Organigramme de décompression LZW. Les variables, OCODE et NCODE (oldcode et newcode), contiennent les codes à 12 bits du fichier compressé, CHAR contient un seul octet, STRING contient une chaîne d'octets.

6 LZ78

L'algorithme LZ78, publié par Abraham Lempel et Jacob Ziv en 1978, est un algorithme de compression sans perte. Il compresse en remplaçant les occurrences répétées de données par des références à un dictionnaire, qui est construit au fur et à mesure de la compression de l'entrée. Étant donné une chaîne d'entrée, l'algorithme la divise en phrases, de sorte que chaque phrase soit l'un des éléments ci-dessus plus un symbole. Par exemple, la chaîne abacbca serait compressée comme suit :

a	b	ac	bc	a
$(0, a)$	$(0, b)$	$(1, c)$	$(2, c)$	$(1, \varepsilon)$
1	2	3	4	5

Figure 11: Exemple de compression avec l'algorithme LZ78

6.1 Compression

Afin de faciliter la mise en œuvre du processus de compression, un dictionnaire D est utilisé avec des entrées de la forme (in, sn) : n, où le couple (in, sn) est la clé et n est la valeur. La paire elle-même est utilisée comme clé, puisque ce dictionnaire permet de vérifier si une paire a déjà été créée, et dans ce cas, il faut obtenir son n. Le dictionnaire qui serait créé pour l'exemple donné au début serait le suivant :

	a	b	ac	bc	a	
$\{$	$(0, a) : 1$	$(0, b) : 2$	$(1, c) : 3$	$(2, c) : 4$	$(1, \varepsilon) : 5$	$\}$

Figure 12: Exemple de dictionnaire pour la compression avec LZ78

6.2 Décompression

Pour décompresser, un dictionnaire W est également utilisé. Ce dictionnaire contiendra une entrée pour chaque paire de compression, mais pour faciliter la décompression, elles ont maintenant la forme n:w. La valeur n est toujours l'index associé à la paire, et w est la chaîne de symboles complète (la phrase) qui représente la paire.