Université Pierre et Marie Curie
DEUST Informatique
Année 2001–2002

Module de mathématiques

Introduction à la compression de données

Table des matières

1 Introduction, historique et quelques techniques de base

1.1 Qu'est ce que la compression de données ?

Définition 1 : *La compression de données est l'opération qui consiste à convertir un flux de données (que l'on appelle aussi flux d'entrée ou données d'origine) en un autre flux de données (que l'on appelle aussi flux de sortie ou flux compressé) qui est de taille plus petite.*

Un flux peut être un fichier sur un disque ou une disquette, un buffer en mémoire ou encore ce qu'envoie un périphérique quelconque. En langage C, ce peut être par exemple `stdin`, `stdout` ou `stderr`.

La compression de données est un art ancien et ne date pas seulement de l'ère de l'ordinateur. Dans la préface de son livre «Data Compression : the complete reference», David Salomon cite par exemple une méthode de compression rudimentaire datant de 1711 : une femme écrivit au Spectator, un journal londonnien, pour demander conseil afin de mieux supporter les longues absences de son mari. Joseph Addison, le conseiller qui lui répondit, lui indiqua comment communiquer avec son mari grâce au «télégraphe sympathique». Cet appareil, conçu par Giovanni Battista Della Porta, un scientifique de la Renaissance, était constitué de deux cadrans circulaires sur le pourtour desquels étaient inscrites les 26 lettres de l'alphabet. Chaque cadran était en outre muni en son centre d'une aiguille magnétisée avec le même aimant. D'après Della Porta, cela devait avoir pour effet de coordonner les aiguilles de telle sorte que lorsqu'une lettre était composée sur un des cadrans, l'aiguille de l'autre cadran devait tourner elle aussi et pointer sur la même lettre. Le chroniqueur ajouta alors qu'en plus des 26 lettres habituelles, les cadrans devraient aussi contenir «quelques mots qui ont toujours leur place dans les lettres passionnées». Le chroniqueur venait de faire de la compression de données : un message tel que «je vous aime» ne demanderait que 3 symboles du cadran au lieu des 10 lettres du message.

Plus récemment, Louis Braille inventa dans les années 1820 le célèbre code qui porte son nom. Ce dernier consiste en des matrices (ou cellules) de 3×2 points mis en relief sur du papier épais. Chacun des six points d'une cellule peut être soit plat soit en relief. Ainsi chaque cellule contient des informations sur 6 bits, et il existe donc 64 cellules différentes. Les lettres, chiffres et ponctuations ne requièrent pas 64 codes, aussi, les cellules restantes sont-elles utilisées pour coder des mots ou des chaînes de caractères souvent utilisés. Évidemment, le taux de compression atteint n'est pas très élevé mais il est tout de même important de compresser des mots en braille parce que les livres écrits dans ce code ont tendance à être très volumineux.

Le Braille permet aux aveugles de communiquer mais pendant plus d'un siècle l'un des codes les plus utilisés dans les communications fut le Morse (développé dans sa version actuelle en 1843 conjointement par Samuel Morse et Alfred Vail). Dans ce code, à chaque lettre, chiffre et ponctuation est associé un ensemble de 1 à 5 traits ou points comme le montre le tableau ci-dessous :

A	·–	J	·–––	T	–	1	·————	.	·–·–·–
B	–···	K	–·–	U	··–	2	··–––	,	––··––
C	–·–·	L	·–··	V	···–	3	···––	:	–––···
Ch	––––	M	––	W	·––	4	····–	?	··––··
D	–··	N	–·	X	–··–	5	·····	’	·––––·
E	·	O	–––	Y	–·––	6	–····	-	–····–
F	··–·	P	·––·	Z	––··	7	––···	–	–··–·
G	––·	Q	––·–			8	–––··	()	–·––·–
H	····	R	·–·			9	––––·	”	·–··–·
I	··	S	···			0	—————		

TAB. 1: Le code Morse

Lorsque l'on envoie du code morse, la durée d'un point est d'une unité de temps, celle d'un trait de trois unités, l'espace entre les points et les traits d'un même caractère est d'une unité. Enfin l'espace entre deux caractères est de trois unités et entre les mots de six unités si l'on transmet en manuel et de cinq si l'on transmet en automatique.

Pourquoi le nombre de traits et de points varie-t-il d'un symbole à l'autre ? Eh bien l'idée est de coder avec le moins de traits/points possible les symboles les plus utilisés. C'est ce que l'on appelle un code de longueur variable. Cela permet de réduire la taille des messages codés, autrement dit, cela permet de compresser le code. Par exemple, j'ai écrit un petit programme calculant le nombre de bits nécessaires au stockage des fichiers LaTeX

du cours de probabilité ainsi qu'au nombre de bits nécessaires à leur stockage en morse. Pour le stockage en morse, j'ai considéré comme un bit une unité de transmission. Ainsi, un «A» est codé sur 8 bits puisque le point requiert une unité, suivie d'une unité d'espace, le trait requiert trois unités et est suivi de trois unités d'espace pour indiquer que c'est la fin du caractère. Le code correspondant au «A» est petit (juste un point et un trait) et il faut déjà 8 bits pour le stocker. On peut donc se dire que les messages en morse doivent être horriblement longs. Et pourtant le résultat du programme sur les fichiers LaTeX est assez parlant :

nom du fichier	taille du fichier d'origine		taille du fichier compressé en morse	
deust.tex	2086	octets	2135	octets
section1.tex	31535	octets	33979	octets
section2.tex	26924	octets	29748	octets
section3.tex	54122	octets	60043	octets
section4.tex	56873	octets	62473	octets
section5.tex	23822	octets	25352	octets
section6.tex	57805	octets	66291	octets
section7.tex	19193	octets	21054	octets
section8.tex	36273	octets	41754	octets
section9.tex	34746	octets	37870	octets
section10.tex	38792	octets	44112	octets
section11.tex	10851	octets	17491	octets
total	393022	octets	442302	octets

Tab. 2: Compression du cours de probabilité par morse

On voit ainsi que la compression obtenue grâce à des lettres comme le «E» ou le «I» sont suffisantes pour obtenir des messages à peine plus longs qu'en ASCII.

Dernier exemple, puisque nous sommes dans un cours de mathématiques, ne trouvez-vous pas que les mathématiques usent et abusent de la compression de données ? «$x \in [1,2]$» ne requiert que 7 symboles et signifie «x appartient à l'intervalle 1,2», phrase qui contient, outre les espaces, 27 caractères. L'utilisation des symboles comme $\in$, $\subset$, $\cap$, $\cup$, bien que plus récente que l'exemple précédent date tout de même de 1894 (c'est Giuseppe Peano qui les introduisit dans son «formulaire de mathématiques»).

Bien évidemment, les codes décrits ci-dessus sont vraiment très rudimentaires et sont largement dépassés en efficacité par les algorithmes de compression actuels. C'est l'apparition des ordinateurs qui a popularisé la compression de données. En effet, les capacités des périphériques de masse ne sont pas infinies et il n'est pas rare de vouloir stocker plus de données que le périphérique ne peut en contenir (des jeux, etc). De plus, depuis quelques années, avec le développement d'internet, il faut pouvoir transférer rapidement de grandes quantités d'informations. La technologie actuelle ne permettant pas d'augmenter énormément les vitesses de transfert des modems, il ne reste plus comme alternative qu'à réduire la taille des données transférées. Ces deux facteurs permettent d'expliquer pourquoi depuis une vingtaine d'années la recherche sur la compression de données a reçu beaucoup de contributions et a évolué très rapidement.

1.2 En quoi consiste un algorithme de compression de données ?

À première vue, les méthodes de compression décrites ci-dessus n'ont pas l'air d'avoir beaucoup de rapports entre elles : Joseph Addison compresse les données en utilisant un seul symbole pour coder des mots souvent utilisés dans les déclarations amoureuses ; le Braille compresse en codant certaines chaînes de caractères grâce à un seul symbole ; le code Morse compresse les données en n'utilisant pas des codes de tailles fixes (comme le code ASCII) mais plutôt des codes de tailles variables. Pourtant, toutes ces méthodes de compression sont fondées sur le même principe :

> Bien qu'ils soient basés sur des idées différentes, qu'ils s'appliquent à des types de données différentes et qu'ils produisent des codes différents, tous les algorithmes de compression de données compressent en éliminant des redondances dans les données d'origine.

L'idée sous-jacente est que si l'on veut réduire la taille d'un fichier, on a intérêt à essayer de réduire «ce

qui revient souvent» dans le fichier. Par exemple, j'ai calculé le nombre de fois où apparaît chaque lettre de l'alphabet dans ceux du cours de probabilité. Voici le résultat :

A	703	B	401	C	499	D	536	E	584	F	172
G	50	H	246	I	307	J	31	K	16	L	431
M	165	N	289	O	355	P	1230	Q	41	R	91
S	344	T	229	U	121	V	126	W	12	X	1362
Y	309	Z	148	a	20892	b	4695	c	9360	d	10879
e	44375	f	3891	g	3423	h	3332	i	21075	j	516
k	222	l	17729	m	8542	n	21091	o	14463	p	8772
q	3111	r	16697	s	20500	t	19606	u	13424	v	3454
w	207	x	2276	y	2824	z	326				

TAB. 3: Nombre d'utilisations des lettres dans le cours de probabilité

Il semble assez évident que si l'on veut réduire la taille des fichiers du cours de probabilité, il vaut mieux essayer de réduire les tailles requises pour stocker les lettres «e» ou «s» plutôt que celles des lettres «K» ou «W». Pourquoi ? Simplement parce que ces lettres apparaissent plus souvent. C'est ce que signifie le terme «redondance» utilisé plus haut.

Le principe du «compresser en éliminant les redondances» permet d'expliquer des phénomènes que vous avez sûrement déjà rencontrés. Par exemple, si vous avez essayé de compresser plusieurs fois un même fichier, vous avez dû vous apercevoir que cela ne sert à rien parce que le nouveau fichier compressé a une taille pratiquement identique à celle du premier fichier compressé et même parfois supérieure. Cela vient du fait que, lorsque vous avez compressé le fichier pour la première fois, vous avez déjà éliminé la plupart des redondances. Il n'en reste donc pratiquement plus à éliminer lorsque vous utilisez l'algorithme de compression sur le fichier compressé. La taille du fichier de sortie peut même augmenter à cause des en-têtes insérés par les logiciels de compression.

fichiers d'origine		fichiers gzippés		gzippés et regzippés		gzippés et bzip2és	
deust.tex	3809	deust.tex.gz	1065	deust.tex.gz.gz	1104	deust.tex.gz.bz2	1360
section1.tex	38142	section1.tex.gz	10999	section1.tex.gz.gz	11041	section1.tex.gz.bz2	11482
section2.tex	33569	section2.tex.gz	9172	section2.tex.gz.gz	9214	section2.tex.gz.bz2	9644
section3.tex	63915	section3.tex.gz	19044	section3.tex.gz.gz	19086	section3.tex.gz.bz2	19511
section4.tex	68099	section4.tex.gz	19225	section4.tex.gz.gz	19267	section4.tex.gz.bz2	19697
section5.tex	27463	section5.tex.gz	9241	section5.tex.gz.gz	9240	section5.tex.gz.bz2	9703
section6.tex	68757	section6.tex.gz	19582	section6.tex.gz.gz	19624	section6.tex.gz.bz2	20041
section7.tex	23096	section7.tex.gz	6969	section7.tex.gz.gz	7011	section7.tex.gz.bz2	7453
section8.tex	43803	section8.tex.gz	11959	section8.tex.gz.gz	12001	section8.tex.gz.bz2	12446
section9.tex	41920	section9.tex.gz	11405	section9.tex.gz.gz	11447	section9.tex.gz.bz2	11877
section10.tex	47436	section10.tex.gz	13527	section10.tex.gz.gz	13570	section10.tex.gz.bz2	13987
section11.tex	13442	section11.tex.gz	3797	section11.tex.gz.gz	3840	section11.tex.gz.bz2	4268
total	473451	total	135985	total	136445	total	141469

TAB. 4: Compression des fichiers du cours de probabilité

Le fait de compresser en éliminant les redondances suggère aussi d'affecter aux symboles ou aux phrases qui reviennent souvent des codes de petites tailles et des codes plus longs à ceux que l'on trouve rarement dans le fichier d'origine. C'est exactement ce principe que suit le code Morse. C'est aussi ce qui est utilisé, comme nous le verrons plus loin, dans le célèbre code de Huffman.

En fait, on peut classer l'ensemble des algorithmes de compression actuels en quatre catégories :

- les RLE (run length encoding), que nous verrons vers la fin de cette section ; ils sont utilisés entre autres dans les algorithmes de compression des modems (comme MNP5) ou bien dans `BinHex`.
- les méthodes statistiques, comme le code de Huffman et le code arithmétique que nous verrons dans la section 2. Celles-ci sont utilisées dans des programmes comme `bzip2` mais aussi dans `MNP5` et `MNP7`, des compresseurs utilisés par les modems, dans les algorithmes de compression utilisés par les fax.
- les méthodes à base de dictionnaires (fondées sur les travaux d'Abraham Lempel et de Jacob Ziv). Ces méthodes sont utilisées dans des logiciels comme `zip`, `gzip` ou encore `compress`. Nous les verrons dans la section 3.

– Les méthodes de compression d'images, de fichiers audio, de films, qui utilisent les limitations de la perception humaine afin d'éliminer des informations peu ou pas perceptibles et obtenir ainsi des taux de compression extrêmement élevés. Nous verrons quelques unes de ces techniques dans la section 4.

Mais insistons bien sur le fait que, même si les idées exploitées par ces trois classes de méthodes diffèrent entre elles, l'idée fondamentale reste toujours l'élimination des redondances.

1.3 Un peu de vocabulaire

Avant d'aller plus loin, nous allons étudier un peu de vocabulaire propre à la discipline de la compression de données.

Définition 2 (compresseur ou encodeur) : *Un* compresseur *ou encore* encodeur *est un logiciel qui compresse les données du flux d'entrée (données d'origine ou données brutes) et qui crée un flux de sortie contenant les données compressées, c'est-à-dire des données avec peu de redondance.*

Définition 3 (décompresseur ou décodeur) : *Un* décompresseur *ou un* décodeur *convertit les données compressées par l'encodeur dans le sens inverse. On dit parfois que le décodeur «expanse» les données.*

Attention : cela ne veut pas dire que l'on retrouvera exactement les données qu'avait l'encodeur sur son flux d'entrée : par exemple, dans la compression d'images, il est fréquent (cf. les fichiers jpeg) d'approximer certaines couleurs afin de réduire la taille du fichier de sortie. On n'a alors aucun moyen de revenir aux couleurs d'origine.

Nous verrons dans la suite de ce cours que le meilleur compresseur n'est pas forcément celui qui réduit au maximum la taille des fichiers en sortie. En effet, il y a aussi un facteur temps qui intervient : on préfèrera souvent un logiciel qui réduit un fichier à 30% de sa taille d'origine en une minute plutôt qu'un logiciel qui le réduira à 29% de sa taille d'origine en une heure. Certaines méthodes, comme celles de la section 2, sont censées faire une première passe sur l'ensemble du flux d'entrée pour analyser les données, puis une deuxième pour réaliser effectivement la compression et créer le flux de sortie. Pour gagner du temps, il existe des variantes de ces méthodes qui analysent le flux d'entrée et créent le flux de sortie en une seule et même passe. Celles-ci sont moins efficaces que les premières en termes de réduction de la taille des données, mais, par contre, elles sont plus rapides. Les premières s'appellent des méthodes semi-adaptatives et les secondes des méthodes adaptatives (car elles adaptent la compression aux données qu'elles sont en train d'analyser). Il existe une troisième catégorie de méthodes : les non adaptatives. Dans celles-ci, on considère que l'analyse des données a été faite une fois pour toutes et est codée «en dur» dans le logiciel de compression. De telles méthodes sont efficaces, à la fois en temps et en compression, lorsque l'on compresse toujours le même type de données.

Définition 4 (méthodes adaptatives, non adaptatives et semi-adaptatives) : *Une méthode de compression* non adaptative *est une méthode qui ne modifie pas ses opérations, ses paramètres, ses tables, en fonction des données particulières qu'elle est en train de compresser.*
Une méthode adaptative étudie les données qu'elle compresse et modifie ses opérations, ses paramètres, en conséquence de manière à atteindre une meilleure compression. Ces modifications se font en même temps que l'on crée le flux de sortie.
Lorsque l'analyse des données qui permet les modifications des paramètres est effectuée dans une première lecture du flux d'entrée, et que l'on relit à nouveau le flux d'entrée afin de créer le flux de sortie, on dit que la méthode est semi-adaptative.

Nous avons vu que la qualité d'un bon algorithme de compression se mesure grâce à deux critères : le temps de compression et la réduction en taille obtenue. Dans certains cas, on peut atteindre des taux de compression extrêmement élevés si l'on consent à perdre quelques informations du flux d'entrée, plus exactement à approximer certaines données d'origine. Évidemment, cela ne peut pas s'appliquer à toutes les données. Par exemple, changer un 'A' par un 'O' dans un texte Word ou dans un programme source peut être catastrophique. Cependant, dans certains cas, de légères pertes d'informations n'ont que très peu de répercussions. Dans une image, changer la couleur d'un pixel ne modifie pas fondamentalement l'image. Il en est de même pour des fichiers audio : éliminer le son que fait un triangle lorsqu'une guitare électrique joue en même temps ne sera pas perçu par beaucoup d'oreilles. On voit donc qu'il existe deux types de compressions, chacune s'appliquant à des types de données bien précises :

Définition 5 (compression avec ou sans perte de données) : *Certaines méthodes perdent quelques informations, i.e., le décompresseur ne peut retrouver exactement les données d'origine. Elles obtiennent de meilleures compressions que les algorithmes sans pertes de données qui, eux, ont des décompresseurs qui retrouvent exactement les données d'origine.*

Il existe plusieurs manières d'envoyer le flux d'entrée au compresseur : on peut envoyer les données en continu, octet par octet (ou par groupe d'octets), c'est ce que l'on appelle le mode d'entrée «par flot» ou encore en anglais «streaming mode», mais on peut aussi séparer le flot d'entrée en blocs de données et compresser séparément ces blocs. C'est ce que l'on appelle le mode «par bloc» ou «block mode» en anglais. Ce mode est utilisé, entre autres, par le fameux algorithme de compression `bzip2`.

Définition 6 (mode d'entrée) : *La plupart des compresseurs utilisent un mode d'entrée par flot, c'est-à-dire que l'encodeur lit un ou plusieurs octets sur le flux d'entrée, les compresse, en relit plusieurs autres, les compresse, et ainsi de suite jusqu'à la fin du fichier. Certains compresseurs toutefois travaillent avec un mode par bloc, c'est-à-dire qu'ils lisent le flux d'entrée bloc par bloc, et qu'ils compressent ces blocs séparément.*

Enfin, nous allons terminer notre tour d'horizon par une des notions les plus importantes en compression, à savoir la mesure de performance d'un algorithme de compression. Comment peut-on comparer deux algorithmes ? L'une des clés est évidemment de comparer les tailles des fichiers de sortie et d'entrée. Cela nous suggère plusieurs critères de comparaison :

Définition 7 (mesures de performance) : *On appelle taux de compression l'expression suivante :*

$$\textit{taux de compression} = 1 - \frac{\textit{taille du fichier de sortie}}{\textit{taille du fichier d'entrée}}.$$

On appelle facteur de compression le ratio suivant :

$$\textit{facteur de compression} = \frac{\textit{taille du fichier d'entrée}}{\textit{taille du fichier de sortie}}.$$

Le taux de compression et le facteur de compression sont de bons indicateurs d'efficacité d'un algorithme. Il existe un autre indicateur qui, lui, mesure la rapidité de l'algorithme, c'est le nombre de cycles par octet, qui correspond au nombre de cycles machine moyen nécessaires pour compresser un octet.

Bien évidemment, on recherchera des algorithmes ayant le plus fort taux de compression possible. Un taux de 0,5 indique par exemple que le fichier de sortie est deux fois plus petit que le fichier d'origine, et un taux de 0,25 indique qu'il est 1/4 de fois plus petit. Par conséquent, plus ce chiffre est élevé, meilleure est la compression. Le facteur de compression correspond à une notion relativement similaire : un facteur de compression de 2 signifie que le fichier de sortie est 2 fois plus petit que le fichier d'entrée.

Afin de comparer les différents algorithmes de compression existant à ce jour, des sites webs ont été créés, qui regroupent un certains nombre de fichiers de tests supposés représentatifs. Les algorithmes de compression sont alors confrontés à ces fichiers, et l'on peut déterminer lequel des algorithmes semble le meilleur pour un type de donnée précis. Parmi ces sites web, on peut citer :

- `http://links.uwaterloo.ca/bragzone.base.html`
- `ftp://nic.funet.fi/pub/graphics/misc/test-images/`
- `http://corpus.canterbury.ac.nz/`

Avant de voir quelques algorithmes «basiques» de compression (cf. la sous-section 1.5), nous allons voir explicitement pourquoi les algorithmes de compression sont «mathématiquement» limités dans leurs performances.

1.4 Limitation des algorithmes de compression

Par limitation, j'entend qu'il ne sert strictement à rien de prendre un fichier d'entrée, de le compresser (sans perte de données), de compresser le fichier résultat, et ainsi de suite de manière à réduire à chaque fois la taille du fichier de sortie. Nous avions vu «intuitivement» que cela n'était pas possible puisque les compressions ne font qu'éliminer des redondances et, qu'à un moment donné, on n'a plus de redondance, donc on ne peut plus

espérer réduire encore la taille du fichier. Nous allons voir dans cette sous-section un argument beaucoup plus mathématique et tout aussi convaincant. Plus exactement, nous allons démontrer la proposition suivante :

Proposition 1 : *Il n'existe pas d'algorithme (sans perte de données) permettant de compresser* **tous** *les fichiers de taille N (en bits). Par compression, on entendra que le fichier résultat a une taille strictement inférieure à la taille du fichier d'entrée, c'est-à-dire N bits.*

Démonstration de la proposition 1 : Supposons qu'un tel algorithme existe. Il y a exactement 2^N fichiers différents de N bits. Puisque notre algorithme arrive à tous les compresser sans perte d'informations, il arrive donc à former 2^N fichiers différents de tailles strictement inférieures à N. Les fichiers sont forcément différents car sinon, à la décompression, on ne pourrait retrouver tous les fichiers que l'on avait en entrée, et l'algorithme de compression ne serait alors pas sans perte de données. On a donc 2^N fichiers différents de tailles $< N$. Comptons maintenant combien il existe de fichiers possibles de tailles inférieures à N : il y a 2^{N-1} fichiers de taille $N-1$ bits. Il y en a 2^{N-2} de taille $N-2$ bits, etc. On peut même considérer que notre algorithme peut potentiellement créer un fichier vide. Il y a donc :

$$2^{N-1} + 2^{N-2} + \cdots + 2 + 1 = 2^N - 1$$

fichiers différents de tailles strictement inférieures à N. Or notre algorithme, s'il existait, en aurait construit 2^N différents. On voit donc qu'un tel algorithme n'existe pas. ⧫

En fait, ce qui se passe c'est que parmi les 2^N fichiers possibles, il y en a toujours un certain nombre qui sont aléatoires, c'est-à-dire qui n'ont absolument aucune redondance. De tels fichiers ne peuvent être compressés. Lorsque l'on a déjà compressé un fichier une fois, on se rapproche de tels fichiers, et il n'existe plus alors d'algorithme pour «faire mieux».

Pour finir cette section d'introduction, nous allons voir quelques uns des premiers algorithmes de compression.

1.5 Quelques algorithmes primitifs de compression

1.5.1 MacWrite

L'algorithme utilisé par le traitement de texte MacWrite pour compresser son texte est relativement primaire et se rapproche un peu de l'idée du Morse : il consiste à coder les caractères les plus souvent employés, à savoir «␣acdefhilnoprst», sur 4 bits. Les autres caractères sont codés grâce à leur code ASCII, donc sur 8 bits, précédés par un caractère d'échappement de 4 bits qui indique que ce qui suit est un code ASCII. Ils sont donc codés sur 12 bits.

L'algorithme de compression traite chaque paragraphe séparément. Il essaye le codage décrit ci-dessus. Si ce codage a une taille inférieur à un codage tout en ASCII, il le garde, sinon l'algorithme écrit tout le paragraphe en ASCII. Afin de différencier un paragraphe écrit en ASCII d'un paragraphe compressé, MacWrite utilise en début de paragraphe un bit de positionnement (flag).

Cet algorithme est relativement primitif mais il permet tout de même d'obtenir une compression appréciable. Par exemple, sur l'ensemble des fichiers LaTeX du cours de probabilités, on obtient un taux de compression de 16%, comme le montre le tableau suivant :

nom du fichier	taille du fichier MacWrite		taille du fichier LaTeX	
deust.tex	3348	octets	3809	octets
section1.tex	31862	octets	38142	octets
section2.tex	29386	octets	33569	octets
section3.tex	53426	octets	63915	octets
section4.tex	56617	octets	68099	octets
section5.tex	21713	octets	27463	octets
section6.tex	57638	octets	68757	octets
section7.tex	18921	octets	23096	octets
section8.tex	37127	octets	43803	octets
section9.tex	35346	octets	41920	octets
section10.tex	40596	octets	47436	octets
section11.tex	13279	octets	13442	octets
total	399259	octets	473451	octets

TAB. 5: compression du cours de probabilités sous MacWrite

1.5.2 Run Length Encoding (RLE)

Une autre idée, là encore relativement primitive, consiste à se dire que si une même donnée d apparaît n fois consécutives sur le flux d'entrée, on peut réaliser une compression en n'écrivant sur le flux de sortie que le couple (n, d). Cette méthode s'appelle RLE car, en anglais, n occurrences consécutives d'une donnée d s'appelle un «run length» de n. Cette technique peut s'appliquer sur des fichiers texte, mais elle n'est pas extrêmement performante pour ce type de données, sauf dans certains textes scientifiques comme des textes mathématiques. Elle s'applique avec un peu plus de succès sur des images.

1.5.3 Run Length Encoding pour des fichiers texte

L'idée du RLE pour les fichiers texte est donc de remplacer les suites de caractères identiques par des couples (taille de la suite,caractère). Par exemple,

n occurrences consécutives d'une donnée d s'appelle un «run length» de n.

devrait s'écrire :

n o(2,c)u(2,r)ences consécutives d'une do(2,n)ée d s'a(2,p)e(2,l)e un (2,<)run length(2,>) de n.

Il faut maintenant trouver une manière de coder tout cela de manière à ne pas confondre les paires avec le texte «normal». L'idée la plus simple est d'utiliser un caractère d'échappement, que l'on notera ici @, pour indiquer que les deux octets qui suivent correspondent respectivement à une taille et au caractère qui se répète. Ainsi, un RLE sur le texte ci-dessus donne :

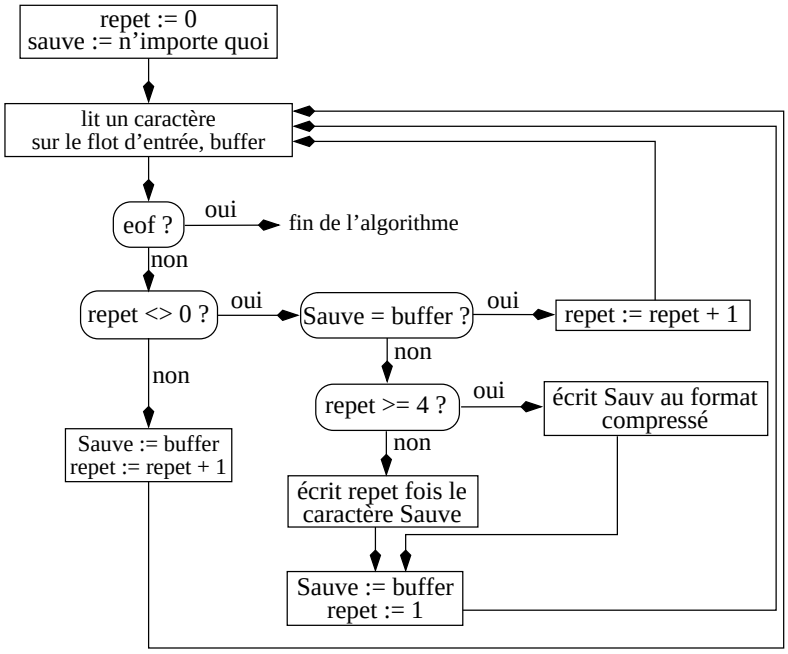
n o@2cu@2rences consécutives d'une do@2née d s'a@2pe@2le un @2<run length@2> de n.

Ici, on s'aperçoit qu'il y a tout de même un problème : on a remplacé les 2 «c» de «occurrences» par la chaîne «@2c» dont la taille est de trois caractères. Autrement dit, on n'a pas réussi à compresser le texte, mais on l'a expansé. C'est tout de même un comble pour un compresseur ! En fait, l'algorithme ne compressera ni n'expansera si la donnée d est répétée trois fois, et il compressera si d est répétée au moins 4 fois, ce qui est assez rare pour les textes courants. Le RLE ne s'applique donc pas à beaucoup de textes. Il pourra être utile néanmoins pour des programmes sources. Par exemple lorsque l'on indente le programme avec des espaces, ou bien dans les commentaires, où l'on peut rencontrer des séquences d'«*» de «=» ou encore de «-». Par exemple, le RLE tel que décrit ci-dessus donne le résultat suivant sur les fichiers du cours de probas (LaTeX est un langage de programmation, et je programme toujours en indentant) :

nom du fichier	taille du fichier RLE		taille du fichier LaTeX	
deust.tex	2606	octets	3809	octets
section1.tex	35936	octets	38142	octets
section2.tex	31776	octets	33569	octets
section3.tex	61934	octets	63915	octets
section4.tex	66071	octets	68099	octets
section5.tex	26560	octets	27463	octets
section6.tex	66368	octets	68757	octets
section7.tex	22153	octets	23096	octets
section8.tex	42219	octets	43803	octets
section9.tex	40389	octets	41920	octets
section10.tex	45683	octets	47436	octets
section11.tex	12947	octets	13442	octets
total	454642	octets	473451	octets

TAB. 6: compression du cours de probabilités avec RLE

Le taux de compression est toutefois relativement faible : sur le cours de probabilités, il est seulement de 3,98%. Vous trouverez ci-dessous l'algorithme de compression correspondant décrit sous forme graphique, et à la fin de cette section le même algorithme écrit en langage C.



L'algorithme de décompression est vraiment trivial : on lit un caractère sur le flux d'entrée. Si ce caractère est l'@, on lit les deux octets suivants, et on sait que le premier de ces octets correspond au nombre de répétitions du second octet. Si l'on n'a pas lu un @, on écrit le caractère lu.

Bien évidemment, lorsque l'on compresse, l'@ peut appartenir au fichier d'origine. Dans ce cas, on devrait écrire sur le flux de sortie un @ qui serait différent de celui signifiant les répétitions de caractères. Le problème est de savoir comment différentier les @. Il existe différentes techniques. Celle utilisée par MNP5 est la suivante : lorsque l'encodeur rencontre sur le flux d'entrée 3 ou plus de 3 octets consécutifs identiques, il écrit trois fois l'octet en question, puis le nombre de répétitions de cet octet. En fait de nombre de répétitions, on écrit plutôt ce nombre - 3. Par exemple, la chaîne «aaaaa» sera codée «aaa2».

En conclusion, le RLE n'est pas une très bonne technique pour compresser des fichiers texte. Par contre, elle a l'avantage d'être très simple et très rapide à mettre en œuvre.

1.5.4 Run Length Encoding pour des images

Si le RLE n'est pas très indiqué pour compresser du texte, il est par contre bien adapté pour la compression d'images. Comme chacun sait, une image est composée de pixels (pour le moment nous supposerons que les pixels ne peuvent prendre que deux valeurs : noir et blanc). Le format le plus naturel pour coder une image est tout simplement de former un tableau, appelé bitmap, contenant les pixels, rangés ligne par ligne, de la gauche vers la droite.

RLE est un «bon» candidat pour la compression d'image parce que, en général, si l'on choisit au hasard un pixel dans l'image, il y a de fortes chances pour que les pixels voisins aient la même couleur. Autrement dit, si l'on examine le bitmap, on risque de voir apparaître de longues séries de 0 ou de 1. Le principe d'un compresseur RLE pour une image bitmap consiste alors à parcourir le bitmap pour rechercher ces suites de bits identiques.

1.5.4.1 Images en noir et blanc

Prenons l'exemple du bitmap 64x54 ci-après.

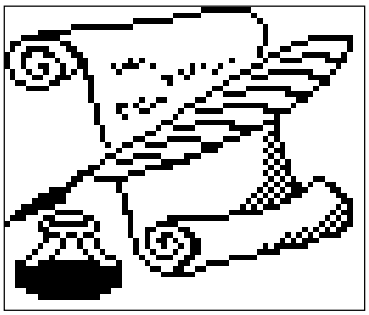


FIG. 1: une image bitmap quelconque

```
                                  #########
                               #####        #
                            #######          #
                      ###########             #
                   ######                     #
                  #######                     #       ########
                ##       ##                   #    ###       #
                #  ####   ##                   #   ##         #
               #  ##  ##   ##                   # ##    #####  #
               #  #     #   ##       # #             #     #  ####    ####
               #  #  ##  #  ##    # ###  #      # #  #     ##            #
                #  #  #  # # #     #          # #   #    ##   ########   #
                #   ##  ###  ##           ##  #       ##  ###       ####
                 ##    ##     #            #          #                ##
                   ####       #                   ##    ########    #
                              #                  #  ####       #####
                              #       #  ##     ##              ##
                               #   ## #  #     ##     ########    #
                               #   #   ##     #  #####       #####
                               #    #        #                #
```

TAB. 7: une portion du bitmap du parchemin en mode texte

```
00000000000000000000000000000000000011111111100000000000000000000
00000000000000000000000000000000111110000000001000000000000000000
00000000000000000000001111111100000000000000010000000000000000000
00000000000011111111111000000000000000000000010000000000000000000
00000011111100000000000000000000000000000000010000000000000000000
00011111110000000000000000000000000000000000010000000111111110000
01100000001100000000000000000000000000000000010000111000000010000
01001111000110000000000000000000000000000000100011000000000010000
10011001100011000000000000000000000000000000101100001111100100000
10010000010001100000001010000000000000001000001001111000011110000
10010011001001100001011100100000010100100000110000000000000010000
01001001001010100000100000000001010001000011000111111110001000000
01000110011100110000000000001100100000001100111000000011110000000
00110000110000010000000000001000000000100000000000000000110000000
00001111000000010000000000000000000011000011111111000010000000000
00000000000000010000000000000000000010011110000000111110000000000
00000000000000010000000010011000001100000000000000011000000000000
00000000000000010001101001000001100000111111110000100000000000000
00000000000000010001000110000010011111000000011111000000000000000
00000000000000010000100000000100000000000000000010000000000000000
```

Tab. 8: une portion du bitmap du parchemin exprimé en bits

La table des bits ci-dessus montre bien qu'il y a de longues suites de 0 et de 1. Donc RLE devrait pouvoir compresser le fichier. Dans la compression de fichiers texte, nous avions vu que le format «compressé» consistait en un triplet d'octets (@, nombre de répétitions, caractère répété). Ici, on peut améliorer le codage. En effet, on peut ne coder que le nombre de répétitions des bits : si l'on remarque, comme sur l'image ci-dessus, que sur la première ligne, il y a 35 «0» consécutifs, suivis par 9 «1» consécutifs, suivis eux-même par 20 «0», on peut très bien coder la première ligne : 35,9,20. En effet, si l'on sait que le premier chiffre correspond à une série de bits «0», forcément le deuxième chiffre correspond à des «1», le troisième à des «0», et ainsi de suite en alternance. Si le premier bit de l'image est égal à «1», on peut dire que l'image commence avec 0 bits «0». C'est le cas de la ligne 9 du bitmap ci-dessus. Cette ligne peut être codée de la manière suivante : «0,1,2,2,2,2,3,2,31,1,1,2,4,5,2,1,3». Afin de pouvoir séparer les différentes lignes, il est judicieux de placer un en-tête dans le fichier de sortie indiquant la taille du bitmap. De plus, il peut être intéressant de rajouter un bit en début de chaque ligne indiquant si l'on écrit la ligne au format compressé ou non. Par exemple la ligne 9 du bitmap du parchemin, lorsqu'elle est écrite au format compressé, est : «0,1,2,2,2,2,3,2,31,1,1,2,4,5,2,1,3», ce qui, si l'on code chaque chiffre sur un octet, fait 17 octets, alors que le bitmap n'ayant que 64 bits en largeur, cette ligne n'était codée que sur 4 octets à l'origine. En fait, le RLE n'est intéressant que pour les «gros» bitmaps, qui ne sont «pas trop» complexes, c'est-à-dire qui n'ont pas trop de détails. Par exemple, lorsqu'on applique RLE sur l'image ci-dessous, provenant du répertoire `/usr/lib/gnomehack/` de ma RedHat 6.0, on passe d'un fichier de 10462 octets (format BMP en noir et blanc) à 27032 octets. En effet, il y a beaucoup trop de détails dans cette image.

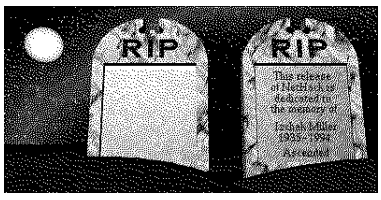


Fig. 2: une image mal compressée par RLE

N'oublions pas que, dans RLE, on écrit le nombre de répétitions de bits identiques. Si l'on code ce nombre sur un octet, RLE n'est valable que si l'on a des répétitions supérieures strictement à 8. Supposons que, dans une image quelconque, p% des séquences soient de longueurs inférieures à 8. Soit m la longueur moyenne de ces séquences. L'image a donc $(100 - p)$% de séquences dont la longueur est supérieure ou égale à 8. Soit M la

longueur moyenne de ces séquences. Alors il est évident que RLE compresse l'image si et seulement si

$$p(m-8)+(100-p)(M-8)>0,$$

ou encore

$$pm+(100-p)M>800.$$

Par exemple, si 30% des séquences ont en moyenne 3 bits, et 70% des séquences ont en moyenne 12 bits, $pm+(100-p)M=30\times 3+70\times 12=930$, RLE compresse l'image. Par contre, si l'image a 40% des séquences avec en moyenne 3 bits, et si 60% des séquences ont en moyenne 9 bits, $pm+(100-p)M=40\times 3+60\times 9=660$, alors RLE expanse plutôt qu'il ne compresse.

Exemple d'utilisation réussie de RLE sur une image : reprenons l'image de l'algorithme RLE page 10. Cette image, au format bitmap (bmp) a une taille de 14654 octets. J'ai compressé cette image avec RLE grâce au programme de la sous-section 1.6.2, ce qui nous fournit alors un fichier de 6060 octets. Voilà un exemple où RLE est très efficace.

On peut atteindre un meilleur taux de compression en ne compressant pas séparément les lignes, mais cela n'est pas vraiment indiqué car si l'on veut modifier l'image, par exemple l'agrandir ou la fusionner avec une autre, on est obligé de décompresser puis recompresser l'image après modification, alors que si l'on code les lignes séparément, il est beaucoup plus facile de modifier l'image (entre autres, sans avoir à réaliser de décompression).

1.5.4.2 Images en teintes de gris

RLE peut aussi servir à compresser des images en teintes de gris. Dans ce cas, les pixels sont encodés sous la forme de couples (nombre de répétitions, intensité de la teinte de gris). Le nombre de répétitions est en principe codé sur un octet. La taille de l'intensité, quant à elle, dépend du nombre de niveaux de gris de l'image d'origine. La plupart du temps, les images en teintes de gris ont 4 ou 8 bits de profondeur.

À l'instar des fichiers texte, il y a peu de chances d'avoir de très longues séquences de pixels ayant exactement la même couleur. Il est donc intéressant de pouvoir coder soit les pixels d'origine, soit les couples (nombre de répétitions, intensité de la teinte de gris). Afin de différencier ces codes, plusieurs techniques sont possibles :

1. On peut rajouter un bit de flag avant chaque code. Si le bit est à 0, les bits qui suivent correspondent au codage de la couleur d'un pixel dans l'image d'origine, sinon ils représente le format compressé. Dans ce cas, pour une image codée sur 8 bits, on a intérêt à utiliser le format compressé dès qu'on a une séquence d'au moins deux teintes identiques consécutives.
2. Une autre technique possible consiste à rajouter devant les codes au format compressé ce qui correspondait à notre @ pour les fichiers texte. Par exemple, si l'image d'origine a entre 128 et 255 teintes de gris, on peut très bien réserver l'octet 255 pour indiquer que ce qui suit est un code au format compressé. Dans ce cas, pour une image codée sur 8 bits, on a intérêt à utiliser le format compressé dès qu'on a une séquence d'au moins quatre teintes identiques consécutives.

Chaque méthode a ses avantages et ses inconvénients. Supposons que, dans une image quelconque ayant 250 teintes de gris, donc une profondeur de 8 bits, p% des teintes soient «isolées», q% soient des séquences de 2 ou 3 teintes identiques (en moyenne, ces séquences sont de taille m) et $100-p-q$% soient des séquences d'au moins 4 teintes identiques (en moyenne elles sont de taille M). Le gain (en bits) de la première méthode est égal à :

$$\frac{\overbrace{p(9-8)}^{\text{9 = 1 bit de flag + 1 octet non compressé (séquence de moins de 2 teintes)}}+q(\overbrace{(1+2\times 8)}^{\substack{\text{1 + 2 × 8 = 1 bit de flag + 1 octet représentant la teinte qui}\\ \text{se répète + 1 octet indiquant le nombre de répétitions}}}-m\times 8)+(100-p-q)((1+2\times 8)-M\times 8)}{100}$$

soit encore

$$\frac{p+q(17-m\times 8)+(100-p-q)(17-M\times 8)}{100}.$$

Lorsque cette valeur est négative, cela signifie que l'encodeur a compressé le fichier d'entrée. Lorsqu'elle est positive, il l'a expansé. Le gain de la deuxième méthode est quant à lui égal à :

$$\frac{p(8-8)+q(m\times 8-m\times 8)+(100-p-q)(3\times 8-M\times 8)}{100}=\frac{(100-p-q)(24-8\times M)}{100}.$$

La première méthode est donc plus intéressante que la deuxième si et seulement si :

$$\frac{p+q(17-m\times 8)+(100-p-q)(17-M\times 8)}{100} - \frac{(100-p-q)(24-8\times M)}{100} \leq 0.$$

ou encore si et seulement si

$$p+q(17-m\times 8)-7\times(100-p-q) = 8p+24q-8qm-700 \leq 0.$$

Exemple 1 : Supposons qu'une image soit constituée de 90% de teintes isolées, 10% de séquences de 2 ou 3 teintes identiques consécutives, et de 10% de séquences d'au moins 4 teintes identiques consécutives. Supposons de plus que parmi les séquences de 2 ou 3 teintes identiques, 40% sont des séquences de longueur 2 et 60% sont des séquences de longueur 3. Alors :

$$m = 40\% \times 2 + 60\% \times 3 = 2,6.$$

Par conséquent,

$$8p+24q-8qm-700 = 8\times 90+24\times 10-8\times 10\times 2,6-700 = 20 \geq 0.$$

Donc le deuxième algorithme de compression RLE est plus efficace que le premier pour compresser cette image.

Supposons maintenant qu'il n'y avait pas 90% de teintes isolées et 10% de séquences de 2 ou 3 teintes identiques consécutives, mais 85% de teintes isolées et 5% de séquences de 2 ou 3 teintes identiques. Alors :

$$8p+24q-8qm-700 = 8\times 85+24\times 5-8\times 5\times 2,6-700 = -4 \leq 0.$$

Donc, dans ce cas, le premier algorithme compresse mieux l'image que le deuxième. ♦

1.5.4.3 Images en couleurs

Dans les images en couleurs, il est assez commun de stocker chaque pixel sous la forme de 3 octets, représentant respectivement l'intensité du pixel dans la couleur rouge, verte et bleue. Si l'on veut utiliser RLE sur des pixels de ce type, il y a très peu de chances que l'on puisse obtenir une compression quelconque. En effet, souvent les pixels voisins ont en commun une ou deux des trois intensités mais rarement les trois. On a donc intérêt à effectuer du RLE sur chacune des couleurs RVB séparément. On se ramène alors à l'algorithme pour les teintes de gris sur chacune de ces couleurs.

1.5.4.4 Balayage de l'image d'origine

Jusqu'à maintenant nous avons toujours effectué des balayages en suivant les lignes du bitmap. En fait, dans le cas de certaines images, cet algorithme n'est pas bien adapté. Considérons en effet une image dans laquelle on n'a que des traits verticaux, comme dans la figure 3. Dans ce cas, RLE ne peut rien compresser du tout, alors que si l'on avait balayé l'image verticalement, on aurait obtenu un très bon taux de compression.

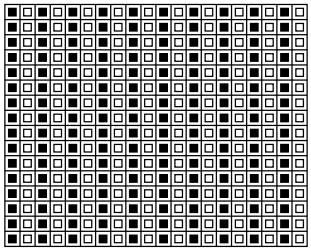


FIG. 3: un bitmap mal compressé par un RLE avec balayage horizontal

De même, les traits auraient pu être en diagonale. Un bon compresseur d'image RLE doit donc balayer l'image horizontalement, verticalement et en zig-zag (cf. figure 4), puis choisir celui des trois balayages qui permet d'atteindre le meilleur taux de compression.

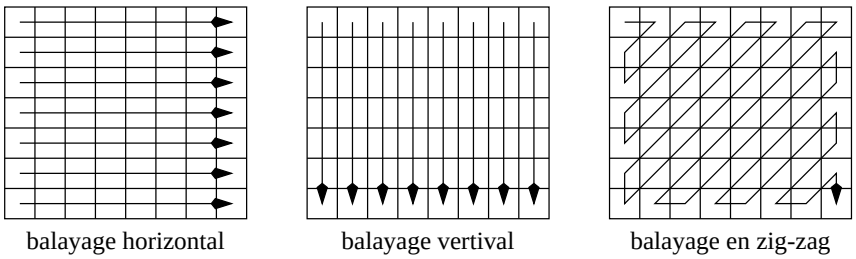


FIG. 4: les différents balayages

Le balayage en zigzag sera décrit en détail dans la section sur JPEG.

1.5.5 BinHex 4.0

BinHex 4.0 est un format de fichier conçu pour Macintosh par Yves Lempereur afin de permettre des transferts de fichiers par réseau sans «pertes de données». Pourquoi perdrait-on des données me direz-vous ? Eh bien, il faut savoir que le code ASCII est un code à 7 bits. Chaque caractère est donc codé grâce à un nombre sur 7 bits. Cependant, le standard ASCII recommende pour des raisons de fiabilité de rajouter un huitième bit de parité. Ainsi, un ordinateur recevant un caractère devrait être en mesure grâce à ce dernier bit de déterminer si ce caractère est bien celui qu'il devait recevoir ou bien s'il a été «corrompu» par un bus ou un périphérique défectueux. Malheureusement, le standard ASCII ne précise pas quand l'on doit obtenir un bit de parité à 0 ou à 1. Aussi, afin d'être plus rapides, en mode ASCII, certains logiciels de transfert ignorent-ils tout simplement le dernier bit et ne transfèrent que les 7 bits du code ASCII. C'est pourquoi l'idée d'Yves Lempereur était de transformer tout fichier, qu'il soit texte ou binaire, en fichier texte (ASCII sur 8 bits dont seulement les sept derniers sont significatifs). Ainsi un fichier BinHex peut être transféré de manière fiable par n'importe quel logiciel de transfert. Le format d'un fichier BinHex 4.0 est le suivant :

1. un premier en-tête indiquant que le fichier en question est bien du BinHex :
   `(This file must be converted with BinHex 4.0)`
2. un deuxième en-tête spécifiant les caractéristiques du fichier :

champ	taille
Longueur (L_1) du nom du fichier (1-63)	1 octet
Nom du fichier	L_1 octets
Version	1 octet
Type	1 long
Créateur	1 long
Flags (And $F800)	1 word
Longueur (L_2) de la zone de donnée	1 long
Longueur (L_3) de la zone de ressource	1 long
CRC	1 word
Zone de donnée	L_2 octets
CRC de la zone de donnée	1 word
Zone de ressource	L_3 octets
CRC de la zone de ressource	1 word

3. le fichier traité par BinHex est alors lu et est compressé par RLE. Le caractère 0x90 est utilisé comme indicateur de format compressé (l'@ que l'on avait utilisé pour les fichiers texte). Le format compressé est : (caractère qui se répète, 0x90, nombre de répétitions). Si 0x90 appartient au fichier d'entrée, il est suivi d'un «0» pour indiquer qu'il s'agit d'un caractère et non d'un marqueur. Exemple :

chaîne d'entrée	chaîne de sortie
0x00 0x12 0x13 0x13 0x12 0xF6	0x00 0x12 0x13 0x13 0x12 0xF6
0x01 0x55 0x55 0x55 0x55 0x55 0x55 0x32	0x01 0x55 0x90 0x06 0x32
0x01 0x55 0x90 0x33	0x01 0x55 0x90 0x00 0x33
0x01 0x90 0x90 0x22 0x90 0x00	0x01 0x90 0x00 0x90 0x00 0x22 0x90 0x00 0x00

4. La chaîne en sortie de l'étape 3 est alors convertie en caractères ASCII (7 bits). Cette chaîne est en fait considérée comme une suite de bits. Ceux-ci sont divisés en groupes de 6 bits, et chaque groupe est alors

converti en l'un des caractères suivants :

```
!"#$}&'()*+,-0123456789@ABCDEFGHIJKLMNOPQRSTUVWXYZ[‘abcdefghijklmnpqr
```

Le fichier de sortie est alors constitué de lignes de 64 caractères obtenus de la manière décrite ci-dessus (sauf la dernière ligne qui peut être plus petite). Chaque ligne est de plus précédée d'un couple de « :».

On voit donc que la seule compression réalisée par BinHex repose sur RLE.

1.6 Algorithmes RLE en langage C

1.6.1 Algorithme RLE pour des fichiers textes

```
/* CG - prog-rle1.c - 27/1/2000 */
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char **argv)
{
  FILE *fic_in, *fic_out;  /* fichiers d'entree et de sortie */
  int  repet=0;            /* compteur de repetition */
  char buffer, sauve;      /* le caractere courant et le dernier caractere lu */
  int  i;                  /* un compteur pour ecrire dans le fichier de sortie */

  /* test des arguments */
  if (argc != 3)
    {
      fprintf(stderr, "usage : rle fic_entree fic_sortie\n");
      exit (EXIT_FAILURE);
    }

  /* premier parametre en entree : le nom du fichier a compresser */
  fic_in = fopen(argv[1],"r");
  if (fic_in == NULL)
    {
      printf("impossible d'ouvrir le fichier %s\n", argv[1]);
      exit(EXIT_FAILURE);
    }

  /* deuxieme parametre en entree : le nom du fichier de sortie, i.e. compresse */
  fic_out = fopen(argv[2],"w");
  if (fic_out == NULL)
    {
      printf("impossible d'ouvrir le fichier  %s\n", argv[2]);
      fclose(fic_in);
      exit(EXIT_FAILURE);
    }

  /* la compression proprement dite */
  /* pour simplifier, on ne lira qu'un caractere a la fois */
  while (fread(&buffer,1,1,fic_in))
    {
      if (repet)
        {
          if (buffer == sauve)
            repet++;
          else
            {
              if (repet >= 4)
                fprintf(fic_out,"@%c%c",(char)repet,sauve);
              else
                for(i=0; i<repet; i++)
                  fprintf(fic_out,"%c",sauve);
              sauve = buffer;
              repet = 1;
            }
        }
      else
        {
          sauve = buffer;
          repet++;
        }
    }

  fclose(fic_in);
  fclose(fic_out);

  return(EXIT_SUCCESS);
```

```
}
```

1.6.2 Algorithme RLE pour des images en noir et blanc

```
/* CG - prog-rle2.c - 30/1/2000 */
#include <stdio.h>
#include <stdlib.h>

/* inclusion du fichier bitmap d'origine au format X11 bitmap */
#include "fig_rle1.xbm"

int main ()
{
           FILE *output;         /* flux de sortie */
           int  i,j,index;       /* index pour parcourir le fichier d'entree */
           char bit_aff;         /* le bit courant dans le fichier d'entree */
           int  current_bit;     /* le dernier bit lu dans le fichier d'origine */
  unsigned char compteur;        /* nombre de repetitions du dernier bit lu */
  unsigned long output_size = 0; /* taille du fichier de sortie */
  unsigned int  taille;          /* pour ecrire la taille du bitmap */

  /* ouverture du fichier de sortie */
  output = fopen("fig_rle1.rle", "w");
  if (output == NULL)
    {
      fprintf(stderr, "impossible d'ouvrir le ficher fig_rle1.rle\n");
      exit(EXIT_FAILURE);
    }

  /* ecriture d'un en-tete : RLE suivi de la taille width * height */
  /* les tailles etant codees chacune sur quatre octets */
  fprintf(output,"RLE");
  taille = rle1_width;
  fwrite(&taille,1,sizeof(int),output);
  taille = rle1_height;
  fwrite(&taille,1,sizeof(int),output);
  output_size += 3 + 2 * sizeof(int);

  /* compression du fichier fig_rle1.xbm */
  /* boucle pour parcourir toutes les lignes */
  for (i=0, index=0; i<rle1_height; i++)
    {
      /* on considere que chaque ligne commence par un certain nombre de blancs */
      current_bit = 0;
      compteur    = 0;
      /* on parcourt tous les bits d'une ligne donnee */
      for (j=0; j<rle1_width; j++)
        {
          /* on recupere le bit numero index dans le bitmap */
          bit_aff = (rle1_bits[index / 8] >> (index % 8)) % 2;

          /* on regarde si ce bit correspond aux bits precedents */
          if (bit_aff == current_bit)
            {
              /* s'il y a trop de bits consecutifs (au moins 256) */
              if (compteur == 255)
                {
                  /* ecriture dans le fichier de sortie du nombre de repetitions */
                  fwrite(&compteur,1,1,output);
                  output_size ++;
                  /* ecriture de 0 bits de l'autre couleur */
                  compteur = 0;
                  fwrite(&compteur,1,1,output);
                  output_size ++;
                  compteur ++;
                }
              compteur++;
            }
          else
            {
              /* ecriture dans le fichier de sortie */
              fwrite(&compteur,1,1,output);
              output_size ++;

              /* on met a jour le bit courant */
              current_bit = bit_aff;
              compteur = 1;
            }
          index++;
```

```
        }
      /* ecriture des derniers bits lus dans la ligne */
      fwrite(&compteur,1,1,output);
      output_size ++;
    }

  /* fermeture de fichier de sortie et affichage de la taille */
  fclose(output);
  printf("taille du fichier de sortie = %ld\n", output_size);

  return(EXIT_SUCCESS);
}
```

1.7 Exercices

Exercice 1 Comme indiqué dans le tableau 6, RLE permet de compresser le fichier `section1.tex` ayant 38142 octets en un fichier `section1.rle` ayant 35936 octets. D'après le tableau 5, ce même fichier est compressé avec MacWrite en un fichier de 31862 octets. Enfin, d'après le tableau 2, le fichier `section2.tex` de 26924 octets (en fait le fichier est un peu plus gros, mais je n'ai comptabilisé que les octets encodables avec le morse) est transformé en morse en un fichier de 29748 octets. Dans ces 3 cas, calculez le taux et le facteur de compression.

Exercice 2 Calculez le taux de compression ainsi que le facteur de compression de la phrase «taux de compression» encodée en Morse.

Exercice 3 Recommencez l'exercice mais en encodant avec MacWrite.

Exercice 4 Compressez avec la version de RLE utilisée par MNP5 la phrase «abaababababbbbaaabbb». Quel est le taux de compression?

Exercice 5 Soit le programme suivant :

```
int  i,j;
char x;

for (i=0; i<26; i++)
  for (j=0; j<10; j++)
    printf("%c", x+j);
```

Calculez le taux de compression obtenu si l'on compresse le flot de sortie avec la version de RLE utilisé par MNP5. Recommencez avec le programme suivant :

```
int  i,j;
char x;

for (j=0; j<10; j++)
  for (i=0; i<26; i++)
    printf("%c", x+j);
```

Exercice 6 Quel est le taux de compression d'une image noir et blanc toute blanche de taille 255×10 compressée avec RLE? Quel est le taux si l'image est toute noire?

Exercice 7 On considère un programme de compression `Comp` qui lit un fichier d'entrée `fic_in` et qui écrit en sortie un fichier compressé `fic_out`. On suppose que les caractères du fichier sont codés sur 3 bits, c'est-à-dire qu'il n'y a que 8 caractères possibles : a_i, $i = 1, \ldots, 8$. Le programme effectue une première passe qui consiste à calculer les fréquences d'apparition des différents caractères, n_i, $i = 1, \ldots, 8$. Dans une deuxième passe, il trie les fréquences par ordre décroissant (les ex-aequos sont triés dans n'importe quel ordre) et affecte au caractère de plus grande fréquence le code binaire «1», au suivant le code «01», et au troisième le code «00», et aux suivants les codes «001», «010», «011», «0110» et «0101». Enfin, le programme écrit `fic_out` à partir de `fic_in` en remplaçant chaque caractère par son code.

1/ Montrez que ce programme compresse toujours strictement le fichier d'entrée.
2/ Est-ce que cela contredit la proposition 1 ? Si vous pensez que la réponse est oui, vous me copierez 100 fois «Non, le prof de maths n'écrit pas n'importe quoi dans son cours», si vous pensez que la réponse est non, vous échappez à la punition ci-dessus, mais vous devez me justifier votre réponse.

Exercice 8 Programmez un algorithme de compression MacWrite. Vous pourrez utiliser avec profit les fichiers `prog-bitio.h` et `prog-bitio.c` qui se trouvent dans le répertoire commun consacré à Visual C++.

2 Les méthodes statistiques

L'idée des méthodes statistiques est d'utiliser des codes de longueurs variables pour encoder les différents octets du flot d'entrée. En fait, elles affectent aux données qui apparaissent souvent dans le flux d'entrée des codes de petites tailles et des codes de tailles plus élevées aux données qui n'interviennent pas «trop» dans le flux d'entrée. Ce type de codage pose deux problèmes principaux :

1. comment affecter aux symboles du flux d'entrée des codes de manière à pouvoir décoder ceux-ci de manière non ambigüe.
2. comment affecter les codes de manière à minimiser la taille du flot de sortie.

Afin de répondre à la deuxième question, nous allons introduire quelques notions de théorie de l'information.

2.1 introduction à la théorie de l'information

Nous savons tous intuitivement ce qu'est une information, mais il nous est difficile d'en donner une définition précise. On sait intuitivement créer une relation d'ordre sur «l'informativité» d'ensembles de données. Par exemple, on sait que l'assertion «quand j'ai lancé cette pièce de monnaie, elle est retombée soit sur pile, soit sur face» fournit moins d'information que «quand j'ai lancé la pièce, elle est retombée sur pile». Mais tout cela reste très intuitif et il semble à première vue difficile de caractériser précisément dans quel cas un message va être plus ou moins informatif qu'un autre. On peut se dire que plus le message est long, plus il contient de l'information, mais ce n'est pas vrai. Par exemple, la phrase «ce pentium 4 tourne à 1,7Ghz» n'est pas plus informative que «ce pentium tourne à 1,7Ghz» car on sait que seuls les pentiums 4 peuvent tourner à cette vitesse. Pire encore, une phrase peut être plus longue et moins informative : «je suis à l'université en $3^{\text{ème}}$ cycle» est moins informatif que «je suis en DEA».

La théorie de l'information, développée par Claude Shannon aux Bell Labs dans les années 40, permet de caractériser mathématiquement cette notion «d'informativité». Plus exactement, elle permet de quantifier, de mesurer l'information et de donner une réponse numérique à la question «quelle quantité d'information est contenue dans telle ou telle donnée». Autrement dit, cette théorie permet de construire une échelle sur «l'informativité» des données. Et c'est particulièrement intéressant car cela va nous permettre de comparer l'informativité exprimée par différents codages. Si deux codes de tailles différentes contiennent la même quantité d'information, le plus petit correspond à une compression du plus grand. La suite de cette sous-section n'est pas fondée directement sur le célèbre article de 1948 de Shannon, «*The Mathematical Theory of communication*», Bell System Technical Journal, **27**, 379–423; 623–656, qui est encore disponible sur la page web des rapports techniques des Bell Labs, mais plutôt sur le livre de Khinchin : «*Mathematical Foundations of Information Theory*», Dover, 1957.

2.1.1 L'entropie de Shannon : une quantification de l'information

Soit $\Omega = \{a_1, a_2, \ldots, a_n\}$ un univers fini et soient $P(\cdot)$ et $Q(\cdot)$ deux lois de probabilité sur cet univers. Rappelons que, par définition, les a_i sont les événements élémentaires. D'après Kolmogoroff, il suffit de déterminer les valeurs de ces deux lois pour les événements a_i pour que celles-ci soient définies sans ambiguïté sur l'ensemble des événements de Ω. Ainsi, dans la suite, nous ne nous servirons que des probabilités des événements élémentaires :

Définition 8 (schéma) : *Soit $\Omega = \{a_1, a_2, \ldots, a_n\}$ un espace probabilisé, de probabilité $P(\cdot)$. Pour tout i, notons $P_i = P(a_i)$. On appelle schéma l'ensemble des a_i, muni des P_i. On le note de la manière suivante :*

$$A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ P_1 & P_2 & \ldots & P_n \end{pmatrix}.$$

Les deux lois $P(\cdot)$ et $Q(\cdot)$ décrivent des états d'incertitude différents sur l'univers Ω. Par exemple, supposons que $\Omega = \{\text{pile,face}\}$ et que $P(\cdot)$ et $Q(\cdot)$ soient tels que :

$$P(\text{pile}) = 0,5 \qquad P(\text{face}) = 0,5 \qquad Q(\text{pile}) = 0,99 \qquad Q(\text{face}) = 0,01.$$

Il est bien évident que le résultat d'un jet d'une pièce régi par la loi $P(\cdot)$ est beaucoup plus incertain que celui d'une pièce régi par la loi $Q(\cdot)$. En effet, dans le premier cas, il y autant de chances de tomber sur pile que sur

face, tandis que dans le second, on est à peu près sûr de tomber sur pile. Il serait donc intéressant de pouvoir quantifier l'incertitude décrite par différents schémas sur Ω. Dans son article de 1948, Claude Shannon proposa comme mesure d'incertitude l'entropie :

Définition 9 (entropie de Shannon) : *Soit* $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ *un schéma. On appelle entropie de Shannon la fonction* $H(\cdot)$ *définie par :*

$$H(P_1, P_2, \dots, P_n) = -\sum_{k=1}^{n} P_k \log P_k,$$

où la base du logarithme est arbitraire, mais fixée une fois pour toutes si l'on compare l'entropie de plusieurs schémas, et où la fonction $p \log p$ *est prolongée par* 0 *en* $p = 0$.

Pourquoi utiliser cette fonction comme mesure d'incertitude ? Eh bien, elle possède un certain nombre de propriétés que l'on aimerait bien retrouver dans une mesure d'incertitude. Voici quelques unes de ces propriétés :

Propriétés de l'entropie de Shannon :

1. $H(P_1, P_2, \dots, P_n) \geq 0$ *pour tout* $\{P_1, \dots, P_n\}$. *De plus,* $H(P_1, P_2, \dots, P_n) = 0$ *si et seulement si un des* P_i *est égal à 1 et tous les autres sont égaux à 0.*
2. *Pour un* n *fixé, l'entropie est maximale si et seulement si* $P_i = 1/n$ *pour tout* i.
3. $H(P_1, \dots, P_n, 0) = H(P_1, \dots, P_n)$.
4. $H(\cdot)$ *est une fonction symmétrique de ses arguments.*
5. *Soient deux schémas* $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ *et* $B = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ Q_1 & Q_2 & \dots & Q_m \end{pmatrix}$ *indépendants, c'est-à-dire que la probabilité que les événements* a_i *se réalisent est indépendante de la réalisation des* b_j *et vice versa. Soit* AB *le schéma correspondant à la réalisation jointe des événements* a_i *et* b_j, *autrement dit :*
$$AB = \begin{pmatrix} a_1 b_1 & \dots & a_i b_j & \dots & a_n b_m \\ P_1 Q_1 & \dots & P_i Q_j & \dots & P_n Q_m \end{pmatrix}.$$
Alors $H(P_1 Q_1, \dots, P_n Q_m) = H(P_1, \dots, P_n) + H(Q_1, \dots, Q_m)$.
6. *Soient deux schémas* $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ *et* $B = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ Q_1 & Q_2 & \dots & Q_m \end{pmatrix}$ *(mutuellement) dépendants. Soit* AB *le schéma correspondant à la réalisation jointe des événements* a_i *et* b_j. *Notons* P_{ij} *la probabilité d'une telle réalisation et notons* R_{ij} *la probabilité que l'événement* b_j *se réalise sachant que l'événement* a_i *s'est réalisé. Alors :*
$$H(P_{11}, \dots, P_{nm}) = H(P_1, \dots, P_n) + \sum_{i=1}^{n} P_i H_i(Q_1, \dots, Q_m),$$
où $H_i(Q_1, \dots, Q_m) = -\sum_{j=1}^{m} R_{ij} \log R_{ij}$.

La propriété 1 montre que l'entropie est nulle si et seulement si l'un des événements a_i est certain, et donc tous les autres sont impossibles. Dans ce cas, il est bien évident qu'il n'y a pas d'incertitude dans le schéma puisque l'on sait quel événement va se réaliser. La propriété 2, quant à elle, indique qu'il n'y a pas de situation plus incertaine que celle où tous les événements élémentaires ont autant de chances de se réaliser les uns que les autres. La propriété 3 précise que si l'on rajoute un événement impossible à l'ensemble des événements élémentaires, cela ne change pas l'incertitude du schéma. La propriété 5 montre que lorsque deux schémas sont indépendants, l'incertitude sur les événements qui vont se réaliser dans les deux schémas est la somme des incertitudes liées à chacun des schémas. Par exemple, supposons que l'on ait deux pièces de monnaie, et que chacune des pièces a autant de chances de tomber sur pile que sur face. Si l'on fait un pari sur le résultat du jet de la première pièce, on a une chance sur deux de se tromper. Si l'on parie sur le résultat des jets des deux

pièces, il y a quatre résultats équiprobables possibles. La deuxième situation est donc plus incertaine que la première. La propriété 6 montre que l'incertitude sur la réalisation jointe de deux événements dépendants est la somme de l'incertitude sur la réalisation du premier événement et de l'incertitude sur le deuxième événement sachant la réalisation du premier.

Démonstration des propriétés ci-dessus :
Propriété 1 : les P_k sont des probabilités, donc ce sont des nombres compris entre 0 et 1. Donc, pour $P_k \neq 0$, $-P_k \log P_k \geq 0$. Puisque $p \log p$ est prolongée par 0 en $p = 0$, pour tout $P_k \in [0,1]$, $-P_k \log P_k \geq 0$ et donc $H(P_1, P_2, \ldots, P_n) \geq 0$ pour tout $\{P_1, \ldots, P_n\}$. Si $P_k \notin \{0,1\}$, alors $P_k \log P_k > 0$. Par conséquent une condition nécessaire pour que $H(P_1, P_2, \ldots, P_n) = 0$ est que les P_k appartiennent à $\{0,1\}$. Or les P_k sont les probabilités des événements élémentaires. Par conséquent, $\sum_{k=1}^{n} P_k = 1$. Donc une condition nécessaire pour que $H(P_1, P_2, \ldots, P_n) = 0$ est que l'un des P_k soit égal à 1 et tous les autres à 0. C'est aussi une condition suffisante puisque $0 \log 0 = 0$ et $1 \log 1 = 0$.
Propriété 2 : On sait que toute fonction convexe φ de $\mathbb{R} \to \mathbb{R}$ vérifie pour tous $x_1, \ldots, x_n$ positifs la propriété suivante :

$$\varphi\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) \leq \frac{1}{n}\sum_{i=1}^{n} \varphi(x_i).$$

Par conséquent, pour $x_i = P_i$ et $\varphi(x) = x \log x$, on obtient :

$$\varphi\left(\frac{1}{n}\right) = \frac{1}{n}\log\frac{1}{n} = \left(\frac{1}{n}\sum_{i=1}^{n} P_i\right)\log\left(\frac{1}{n}\sum_{i=1}^{n} P_i\right) \leq \frac{1}{n}\sum_{i=1}^{n} P_i \log P_i = -\frac{1}{n} H(P_1, P_2, \ldots, P_n).$$

Par conséquent,

$$H(P_1, P_2, \ldots, P_n) \leq \log n = H\left(\frac{1}{n}, \ldots, \frac{1}{n}\right).$$

Propriété 3 : $H(P_1, \ldots, P_n, 0) = -\sum_{i=1}^{n} P_i \log P_i - 0 \log 0 = -\sum_{i=1}^{n} P_i \log P_i = H(P_1, \ldots, P_n)$.
Propriété 4 : évident vu la définition de l'entropie.
Propriété 5 : $H(P_1Q_1, \ldots, P_nQ_m) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P_iQ_j \log(P_iQ_j) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P_iQ_j(\log P_i + \log Q_j)$
$= -\sum_{i=1}^{n} P_i \log P_i \sum_{j=1}^{m} Q_j - \sum_{j=1}^{m} Q_j \log Q_j \sum_{i=1}^{n} P_i$. Or on sait que $\sum_{i=1}^{n} P_i = \sum_{j=1}^{m} Q_j = 1$. Par conséquent,

$$H(P_1Q_1, \ldots, P_nQ_m) = -\sum_{i=1}^{n} P_i \log P_i - \sum_{j=1}^{m} Q_j \log Q_j = H(P_1, \ldots, P_n) + H(Q_1, \ldots, Q_m).$$

Propriété 6 : P_{ij} est la probabilité que les événements a_i et b_j se réalisent. Autrement dit, $P_{ij} = P(a_i, b_j)$. Or, on sait, d'après la formule bien connue du cours de probabilité, que $P(a_i, b_j) = P(a_i) \times P(b_j|a_i)$. Notons $R_{ij} = P(b_j|a_i)$. Alors $P_{ij} = P_iR_{ij}$. Donc

$$\begin{aligned} H(P_{11}, \ldots, P_{nm}) &= -\sum_{i=1}^{n}\sum_{j=1}^{m} P_{ij} \log(P_{ij}) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P_iR_{ij}(\log P_i + \log R_{ij}) \\ &= -\sum_{i=1}^{n} P_i \log P_i \sum_{j=1}^{m} R_{ij} - \sum_{i=1}^{n} P_i \sum_{j=1}^{m} R_{ij} \log R_{ij}. \end{aligned}$$

Or, $\sum_{j=1}^{m} R_{ij} = \sum_{j=1}^{m} P(b_j|a_i) = 1$ quel que soit i et $-\sum_{j=1}^{m} R_{ij} \log R_{ij} = H_i(Q_1, \ldots, Q_m)$. Donc

$$H(P_{11}, \ldots, P_{nm}) = -\sum_{i=1}^{n} P_i \log P_i + \sum_{i=1}^{n} P_i H_i(Q_1, \ldots, Q_m) = H(P_1, \ldots, P_n) + \sum_{i=1}^{n} P_i H_i(Q_1, \ldots, Q_m).$$

♦

Lorsque l'on réalise une expérience (aléatoire), on part d'un schéma contenant de l'incertitude (un certain nombre d'événements peuvent se réaliser, mais on ne sait pas lequel cela va être) et, une fois l'expérience terminée, on sait quel événement s'est réalisé. Par conséquent, la réalisation de l'expérience a permis d'obtenir des *informations* et l'incertitude du schéma est complètement éliminée. Ainsi, on peut dire que l'information qui nous est fournie par le résultat d'une expérience aléatoire sur un schéma consiste à éliminer l'incertitude qui existait avant que l'expérience ne soit réalisée. Par conséquent, plus l'incertitude sur le schéma était grande, plus la quantité d'information obtenue pour éliminer cette incertitude est grande. Si l'on représente l'incertitude sur un schéma $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ par $H(P_1, \dots, P_n)$, alors il est naturel d'exprimer la quantité d'information apportée par l'élimination de cette incertitude par une fonction croissante de $H(P_1, \dots, P_n)$. En fait, il est pratique de prendre une fonction linéaire. En effet, si $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ et $B = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ Q_1 & Q_2 & \dots & Q_m \end{pmatrix}$ sont des schémas indépendants, on a vu que $H(P_1Q_1, \dots, P_nQ_m) = H(P_1, \dots, P_n) + H(Q_1, \dots, Q_m)$. Autrement dit, l'incertitude sur le schéma AB est la somme des incertitudes des schémas A et B. Il est donc logique de considérer que l'information nécessaire pour éliminer l'incertitude sur le schéma AB est la somme des informations pour éliminer l'incertitude de A et pour éliminer celle de B.

2.1.2 Unicité de la quantification d'information

Nous venons de voir dans la sous-section précédente que l'entropie possède un certain nombre de propriétés que l'on aimerait retrouver dans une mesure d'information. Nous allons voir maintenant qu'en fait c'est la seule fonction qui vérifie ces propriétés. Plus exactement, nous allons démontrer le théorème suivant :

Théorème 1 (unicité des mesures d'information) : *Soit $H(P_1, \dots, P_n)$ une fonction définie pour tout entier n et pour tous réels positifs ou nuls $P_1, \dots, P_n$ tels que $\sum_{i=1}^n P_i = 1$. Si, pour tout n, cette fonction est continue par rapport à ses arguments et si elle vérifie les trois propriétés suivantes :*

1. *Pour n fixé et pour $P_1, \dots, P_n$ tels que $\sum_{i=1}^n P_i = 1$, la valeur maximale de $H(P_1, \dots, P_n)$ est atteinte lorsque $P_i = 1/n$ pour tout $i \in \{1, \dots, n\}$.*
2. *Soient $A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ P_1 & P_2 & \dots & P_n \end{pmatrix}$ et $B = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ Q_1 & Q_2 & \dots & Q_m \end{pmatrix}$ deux schémas, et soient R_{ij} la probabilité que l'événement b_j se réalise sachant que l'événement a_i s'est réalisé et P_{ij} la probabilité jointe de a_i et b_j. Alors :*

$$H(P_{11}, \dots, P_{nm}) = H(P_1, \dots, P_n) + H_A(Q_1, \dots, Q_m), \text{ où } H_A(Q_1, \dots, Q_m) = \sum_{i=1}^n P_i H(R_{i1}, \dots, R_{im}).$$

3. *$H(P_1, \dots, P_n, 0) = H(P_1, \dots, P_n)$.*

Alors $H(P_1, \dots, P_n) = -\lambda \sum_{i=1}^n P_i \log P_i$, où λ est une constante positive ou nulle.

Démonstration du théorème 1 : Afin de simplifier les notations de la démonstration, nous allons poser :

$$L(n) = H\left(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}\right).$$

La démonstration va être effectuée en deux parties : premièrement, nous allons montrer que $L(n) = \lambda \log n$, qui est le cas particulier de $H(P_1, \dots, P_n) = -\lambda \sum_{i=1}^n P_i \log P_i$ pour tous les P_i égaux à $1/n$; dans un deuxième temps, nous allons généraliser ce résultat à n'importe quels P_i.

Première partie : $L(n) = \lambda \log n$

D'après les propriétés 1 et 3 du théorème, on a forcément :

$$L(n) = H\left(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}, 0\right) \leq H\left(\frac{1}{n+1}, \frac{1}{n+1}, \dots, \frac{1}{n+1}\right) = L(n+1).$$

Par conséquent, $L(n)$ est une fonction non décroissante de n. Considérons maintenant deux entiers positifs m et r, ainsi que m schémas finis mutuellement indépendants $S_1, \ldots, S_m$ tels que chacun des schémas contient r événements équiprobables. Notons

$$H(S_k) = H\left(\frac{1}{r}, \frac{1}{r}, \ldots, \frac{1}{r}\right)$$

l'entropie de chaque schéma S_k. On sait que $H(S_k) = L(r)$. D'après la propriété 2 du théorème, lorsque deux schémas $A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ P_1 & P_2 & \ldots & P_n \end{pmatrix}$ et $B = \begin{pmatrix} b_1 & b_2 & \ldots & b_m \\ Q_1 & Q_2 & \ldots & Q_m \end{pmatrix}$ sont indépendants, on a $H(P_1Q_1, \ldots, P_nQ_m)$ $= H(P_1, \ldots, P_n) + H(Q_1, \ldots, Q_m)$ car les R_{ij} de la propriété 2 sont, dans ce cas, égaux aux Q_j. Par récurrence, on obtient donc :

$$H(S_1 S_2 \ldots S_m) = \sum_{i=1}^{m} H(S_i) = mL(r).$$

Or, le produit des schémas $S_1, \ldots, S_m$ correspond à un schéma ayant r^m événements équiprobables. Par conséquent, son entropie est égale à $L(r^m)$. Donc :

$$L(r^m) = mL(r). \tag{1}$$

De la même manière, pour tout autre couple d'entiers positif n et s,

$$L(s^n) = nL(s). \tag{2}$$

Maintenant, choisissons les nombres n, r et s arbitrairement, et sélectionnons m de telle manière que m soit déterminé par les inégalités suivantes :

$$r^m \leq s^n \leq r^{m+1}. \tag{3}$$

Alors :

$$\begin{gathered} m \log r \leq n \log s \leq (m+1) \log r, \\ \frac{m}{n} \leq \frac{\log s}{\log r} \leq \frac{m}{n} + \frac{1}{n}. \end{gathered} \tag{4}$$

D'après l'équation (3) et la monotonie de la fonction $L(n)$, il résulte que :

$$L(r^m) \leq L(s^n) \leq L(r^{m+1}).$$

Donc, d'après les équations (1) et (2),

$$mL(r) \leq nL(s) \leq (m+1)L(r),$$

de telle sorte que :

$$\frac{m}{n} \leq \frac{L(s)}{L(r)} \leq \frac{m}{n} + \frac{1}{n}. \tag{5}$$

Donc, d'après les équations (4) et (5),

$$\left| \frac{L(s)}{L(r)} - \frac{\log s}{\log r} \right| \leq \frac{1}{n}.$$

Or, la partie gauche de l'inégalité est indépendante de n (nous avons choisi r et s arbitrairement), et n peut être choisi arbitrairement grand. Par conséquent, en faisant tendre n vers $+\infty$, on obtient :

$$\frac{L(s)}{L(r)} = \frac{\log s}{\log r}.$$

Puisque r et s sont arbitraires, il s'ensuit que, pour tout x, $L(x) = \lambda \log x$, où λ est une constante arbitraire. D'après la monotonie de la fonction $L(n)$, on a forcément $\lambda \geq 0$, ce qui conclut la première partie de la démonstration.

Deuxième partie : Le cas général

Considérons maintenant des nombres rationnels P_k, $k \in \{1, \ldots, n\}$. Posons :

$$P_k = \frac{g_k}{g}, \ \forall\, k \in \{1, \ldots, n\},$$

où les g_k sont des entiers positifs ou nuls et où $g = \sum_{k=1}^{n} g_k$. Soit $A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ P_1 & P_2 & \ldots & P_n \end{pmatrix}$ un schéma fini. Le problème auquel on est confronté est de définir une entropie sur A. Pour cela, considérons un schéma B, dépendant de A, défini de la manière suivante : B contient g événements $b_1, \ldots, b_g$. Les b_i sont alors regroupés en n groupes contenant respectivement $g_1, \ldots, g_n$ événements. Si a_k se réalise dans le schéma A, alors, dans le schéma B, tous les g_k événements du $k^{\text{ème}}$ groupe ont une probabilité $1/g_k$, et tous les événements des autres groupes ont une probabilité égale à 0.

Par exemple, si $A = \begin{pmatrix} a_1 & a_2 & a_3 \\ 1/6 & 2/6 & 3/6 \end{pmatrix}$, alors $B = \begin{pmatrix} b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \\ Q_1 & Q_2 & Q_3 & Q_4 & Q_5 & Q_6 \end{pmatrix}$. Le groupe 1 est constitué de $\{b_1\}$, le groupe 2 de $\{b_2, b_3\}$ et le groupe 3 de $\{b_4, b_5, b_6\}$. De plus,

$$(Q_1, Q_2, Q_3, Q_4, Q_5, Q_6) = \begin{cases} (1,0,0,0,0,0) & \text{si } a_1 \text{ se réalise,} \\ \left(0, \frac{1}{2}, \frac{1}{2}, 0, 0, 0\right) & \text{si } a_2 \text{ se réalise,} \\ \left(0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right) & \text{si } a_3 \text{ se réalise.} \end{cases}$$

Lorsqu'un événement a_k se réalise dans le schéma A, le schéma B se réduit à un système de g_k événements équiprobables (ceux du $k^{\text{ème}}$ groupe). Donc, l'entropie conditionnelle

$$H_k(B) = H\left(\frac{1}{g_k}, \frac{1}{g_k}, \ldots, \frac{1}{g_k}\right) = L(g_k) = \lambda \log g_k,$$

et, d'après la propriété 2 du théorème,

$$H_A(B) = \sum_{k=1}^{n} P_k H_k(B) = \lambda \sum_{k=1}^{n} P_k \log g_k = \lambda \sum_{k=1}^{n} P_k \log P_k + \lambda \log g. \tag{6}$$

Revenons au schéma produit AB. Celui-ci est constitué d'événements $a_k b_l$, $1 \le k \le n$, $1 \le l \le g$. Un tel événement est réalisable si et seulement si b_l appartient au $k^{\text{ème}}$ groupe. Par conséquent, pour un k donné, le nombre d'événements $a_k b_l$ réalisables est g_k. Donc le nombre total d'événements réalisables dans le schéma AB est égal à $\sum_{k=1}^{n} g_k = g$. La probabilité de chaque événement $a_k b_l$ est égale à $P_k/g_k = 1/g$. Ainsi, AB est constitué d'événements impossibles (probabilité = 0) et d'événements équiprobables. Par conséquent, d'après la propriété 3 du théorème et la première partie de la démonstration,

$$H(AB) = L(g) = \lambda \log g.$$

D'après la propriété 2 et l'équation (6),

$$\lambda \log g = H(A) + \lambda \sum_{k=1}^{n} P_k \log P_k + \lambda \log g,$$

et donc

$$H(A) = H(P_1, \ldots, P_n) = -\lambda \sum_{k=1}^{n} P_k \log P_k. \tag{7}$$

Pour conclure, l'équation (7) n'a été montrée que pour des P_k rationnels, mais elle reste valable pour n'importe quels nombres réels d'après l'hypothèse de continuité de la fonction $H(\cdot)$. ♦

2.1.3 Application au codage

Commençons par un exemple simple : on va jouer à pile ou face. On sait qu'avant de lancer la pièce, celle-ci peut retomber sur pile ou face. On doit lancer la pièce pour éliminer cette incertitude. Le résultat du jet peut être exprimé à l'aide des mots pile et face, oui et non, ou encore des bits 0 et 1. Donc un simple bit suffit pour représenter l'incertitude sur la face sur laquelle va retomber la pièce. Cet exemple est relativement trivial, mais ce qui est important c'est qu'il peut être aisément généralisé à d'autres situations. En fait, beaucoup de problèmes pratiques peuvent s'exprimer à l'aide d'un certain nombre de bits. Le problème réside alors dans la recherche du nombre minimal de bits (ou de questions oui/non) pour exprimer le problème.

Deuxième exemple : on a un jeu de 64 cartes. Pour simplifier, appelons les cartes par un numéro plutôt que par leur vrai nom (valet de pique,...). Je tire une carte au hasard et je vous demande de trouver quelle est cette carte, autrement dit de trouver à quel nombre elle correspond. Notre problème, tel que nous l'avons formulé dans le paragraphe précédent est de trouver le nombre minimum de questions oui/non qu'il faut poser pour être sûr de localiser la carte en question. On sait bien que pour trouver le plus rapidement possible un nombre dans un intervalle, la méthode la plus rapide est une dichotomie. Autrement dit, on va poser une première question : «est-ce que le nombre est compris entre 1 et 32», puis en fonction du résultat, «est-ce que le nombre est compris entre 1 et 16» ou «entre 33 et 48», et ainsi de suite. Comme on sépare toujours l'intervalle en 2, le nombre de questions nécessaires pour trouver à coup sûr la carte est le plus petit nombre entier N tel que $2^N \geq 64$. Donc $N = \lceil \log_2 64 \rceil = 6$. Par conséquent, 6 est le nombre minimum de bits dont on a besoin pour résoudre le problème.

Passons maintenant à un problème plus pratique. Prenons un transmetteur qui envoie des données sur une ligne de communication, un modem par exemple. Pour être plus général, on supposera que les données transmises ne le sont pas en binaire, mais qu'elles le sont dans un alphabet à n symboles, $\{a_1, a_2, \ldots, a_n\}$. On peut donc coder chaque symbole sur un digit en base n, ce qui signifie qu'on utilise $\log_2 n$ bits pour coder les symboles. Supposons que le transmetteur envoie s symboles par unité de temps. Puisque l'on a vu que chaque symbole était équivalent à $\log_2 n$ bits, on peut en conclure que le transmetteur envoie $s \log_2 n$ bits d'information par unité de temps. Autrement dit,

$$H = s \log_2 n \text{ est la quantité d'information transmise par unité de temps.}$$

Supposons maintenant que chaque symbole a_i appararaisse dans le message transmis avec une probabilité P_i. Si toutes les probabilités des symboles sont égales, $P_i = P = 1/n$. Par conséquent, puisque $H = s \log_2 n$, on peut exprimer H de la manière suivante : $H = s \log_2(1/P) = -s \log_2 P = -s \sum_{i=1}^{n} P_i \log_2 P_i$. La quantité d'information transmise par unité de temps est donc égale à moins s fois la somme des $P_i \log_2 P_i$. On comprend donc que $-sP_i \log_2 P_i$ est la contribution de chaque symbole à la quantité d'information transmise par unité de temps. On retrouve ici l'entropie que nous avions vu dans les sous-sections précédentes. Cette formule se généralise bien évidemment au cas où les P_i ne sont pas tous identiques puisque H représente la quantité d'information transmise par unité de temps :

$$H = \sum_{i=1}^{n} H_i = -s \sum_{i=1}^{n} P_i \log_2 P_i.$$

La quantité moyenne d'information transmise par symbole (un digit en base n) est donc égale à H/s, ou encore $-\sum_{i=1}^{n} P_i \log_2 P_i$. Par analogie l'entropie d'un seul symbole a_i est $-P_i \log_2 P_i$. Elle correspond à l'information apportée en moyenne par a_i lorsque l'on transmet un symbole au hasard. La place à réserver pour a_i est alors égale à $-nP_i \log_2 P_i$ bits. On voit déjà se profiler un codage des symboles avec des tailles variables puisqu'il n'y a aucune raison que les $-P_i \log_2 P_i$ soient tous égaux entre eux.

Nous avions vu dans les sous-sections précédentes que l'entropie était maximale lorsque tous les P_i étaient identiques. Autrement dit, le nombre de bits nécessaires pour coder un message est plus grand quand, en moyenne, tous les symboles apparaissent autant de fois les uns que les autres dans le message. Un tel message ne contient en principe que des données aléatoires. On remarque qu'il ne peut être compressé : en effet les entropies de tous les symboles sont égales à $\frac{\log_2 n}{n}$, et donc la taille minimale pour coder chaque symbole est $\log_2 n$, ce qui était déjà la taille de son codage d'origine.

Afin de savoir si un codage des symboles est optimal, on peut définir la redondance dans les données :

Définition 10 (redondance dans les données) : *Soit un alphabet $\{a_1, \ldots, a_n\}$. Soit un code sur un message ou une donnée tel que la longueur des codes des a_i est k_i. La redondance R de ce message est égale à la différence entre le nombre de bits utilisés en moyenne pour coder un symbole du message et le nombre moyen de bits nécessaires (autrement dit l'entropie du message) :*

$$R = \sum_{i=1}^{n} P_i k_i - \sum_{i=1}^{n} P_i \log_2 P_i.$$

Bien évidemment, lorsque $R = 0$, les données sont compressées au maximum puisque le code n'utilise pas plus de bits que ne le requiert l'entropie, et que l'entropie correspond au nombre de bits minimum nécessaires pour stocker le message.

Exemple 2 : Considérons un alphabet contenant 4 symboles, a_1, a_2, a_3, a_4, dont les probabilités d'apparition sont toutes égales à 0,25. Si l'on veut coder tout l'alphabet, alors chaque lettre a_i doit être codée, comme nous avons vu plus haut, grâce à $-nP_i \log_2 P_i = -\log_2 0,25 = 2$ bits. On peut donc utiliser le codage suivant :

$$a_1 \equiv 00 \qquad a_2 \equiv 01 \qquad a_3 \equiv 10 \qquad a_4 \equiv 11.$$

Ce codage est optimal dans le sens où l'on a bien $-\sum_{i=1}^{n} P_i log_2 P_i = 2 = \sum_{i=1}^{n} P_i \times 2$.

Évidemment, dans la pratique il est rare que chaque symbole ait la même probabilité d'apparition (cf. la table 3, page 5). Considérons donc un alphabet de 4 symboles, a_1, a_2, a_3, a_4, dont les probabilités d'apparition sont respectivement 0,4, 0,25, 0,25 et 0,1. Dans ce cas, l'entropie de l'alphabet, et donc le nombre de bits moyen nécessaires pour coder un symbole est : $-\sum_{i=1}^{n} P_i \log_2 P_i = -0,4 \log_2 0,4 - 0,25 \log_2 0,25 - 0,25 \log_2 0,25 - 0,1 \log_2 0,1 \approx 1,861$. Donc, en moyenne, il faudrait que chaque symbole soit codé sur 1,861 bits pour obtenir une compression optimale. Codons les symboles respectivement 00, 01, 10 et 11. Dans ce cas, la redondance R est égale à $2 \times (0,4 + 0,25 + 0,25 + 0,1) - 1,861 = 0,139$. Ceci signifie qu'en moyenne, lorsqu'on écrit un symbole, on dépense 0,139 bits de trop. Il devrait donc exister d'autres codages permettant de ne pas dépenser ces bits superflus. ♦

Considérons maintenant le codage suivant :

symbole	proba	code 1	code 2
a_1	0,4	1	1
a_2	0,25	01	01
a_3	0,25	010	000
a_4	0,1	001	001

Quand des données sont transmises avec les codes 1 et 2, le nombre moyen de bits transmis par symbole est alors : $0,4 \times 1 + 0,25 \times 2 + 0,25 \times 3 + 0,1 \times 3 = 1,95$. On utilise donc en moyenne 1,95 bits pour transmettre un symbole. Ce n'est pas encore optimal puisqu'en théorie les symboles pourraient être transmis en moyenne grâce à 1,861 bits, mais il y a déjà une amélioration par rapport à l'exemple ci-dessus. Illustrons maintenant l'utilisation du code 1 sur une séquence de 20 symboles :

$$a_1 a_3 a_2 a_1 a_3 a_3 a_4 a_2 a_1 a_1 a_2 a_2 a_1 a_4 a_1 a_3 a_3 a_1 a_2 a_1$$

Cette séquence sera codée :

$$1|010|01|1|010|010|001|01|1|1|01|01|1|001|1|010|010|1|01|1$$

Notons que cette séquence a 39 bits. Par conséquent, en moyenne, les symboles sont codés sur $39/20 = 1,95$ bits. Avec le code 2, nous aurions bien évidemment obtenu le même résultat. Décodons maintenant la chaîne de bits que nous avons écrite. Le premier bit commence par un «1», donc c'est forcément a_1. Le deuxième bit est un «0», cela correspond donc à un des symboles a_2, a_3 ou a_4. Il est suivi d'un «1», donc le deuxième symbole ne peut être que a_2 ou a_3. Si c'est a_2, il est suivi par les bits «001», qui correspondent au symbole a_4, par contre si c'est a_3, il est suivi par les bits «01» qui correspondent au symbole a_2. Donc le code 1 est ambigü car il ne permet pas de différencier les séquences $a_2 a_4$ et $a_3 a_2$.

Au contraire, le code 2, lui, permet de décoder sans ambiguité une séquence codée. Par exemple, celle du paragraphe précédent s'exprime sous la forme :

$$1|000|01|1|000|000|001|01|1|1|01|01|1|001|1|000|000|1|01|1$$

Le premier bit est un «1», il s'agit donc d'un a_1. Le deuxième bit est un «0», donc c'est un a_2, a_3 ou a_4. Le bit suivant est encore un «0», donc seuls a_3 et a_4 restent possibles. Le bit suivant est un «0», il ne reste donc qu'une seule possibilité : a_4. Les bits suivants sont «01», là encore, une seule possibilité : a_2, et ainsi de suite. Le code 2 est donc non ambigü et est par conséquent un bon candidat pour compresser des données.

Problème : quelle propriété nous permettrait de différencier un code ambigü d'un code non ambigü ? Cette propriété existe et s'appelle la propriété du préfixe :

Définition 11 (propriété du préfixe) : *Cette propriété dit que si une séquence de bits a été affectée au code d'un symbole, alors aucun autre code ne devrait commencer par cette séquence. Autrement dit, cette séquence ne doit être le préfixe d'aucun autre code. Les codes qui vérifient cette propriété sont non ambigüs.*

Le code 1 ne respectait pas cette propriété puisque le code du symbole a_2 était le préfixe du code de a_3.

La conception d'un codage à taille variable doit donc satisfaire aux deux exigences suivantes :

1. affecter des codes de faibles tailles aux symboles qui apparaissent fréquemment,
2. respecter la propriété du préfixe.

Ces deux exigences ne suffisent pas encore pour obtenir les codes les plus courts, c'est-à-dire avec un minimum de redondance. On a encore besoin d'un algorithme performant pour parvenir à cela. Cet algorithme prendra en entrée les probabilités (fréquences) d'apparition des symboles et renverra les codes correspondants. Il existe au moins deux algorithmes «partiellement» optimaux : celui de Shannon-Fano et celui de Huffman.

2.2 Le codage de Huffman

L'algorithme de Huffman est très utilisé en compression de données et, depuis son développement en 1952, il a suscité beaucoup de recherches. Il sert de base à de nombreux programmes. JPEG s'en sert par exemple dans une de ses étapes de compression.

Principe du codage de Huffman : L'algorithme consiste :

1. à dresser une liste des symboles de l'alphabet triée par ordre décroissant de fréquence d'apparition des symboles dans le texte.
2. à construire un arbre, dans lequel chaque symbole se trouve sur une feuille, de la manière suivante : (i) on part d'un arbre vide ; (ii) on prend les deux symboles de la liste ayant les plus petites fréquences, appelons-les a_i et a_j, et on les enleve de la liste ; (iii) on crée un symbole représentant la réunion des deux choisis, appelons-le a_{ij}, et on le rajoute à la liste ; (iv) on rajoute alors à l'arbre un arbre binaire constitué d'une racine, a_{ij}, et de deux feuilles, a_i et a_j ; (v) on itère à partir de l'étape (ii) jusqu'à ce qu'il ne reste plus qu'un seul symbole.
3. à parcourir l'arbre ainsi créé afin de déterminer le codage de chaque symbole de l'alphabet.

Exemple 3 : considérons cinq symboles, a_1, a_2, a_3, a_4 et a_5 apparaissant dans un texte avec les fréquences (probabilités) suivantes :

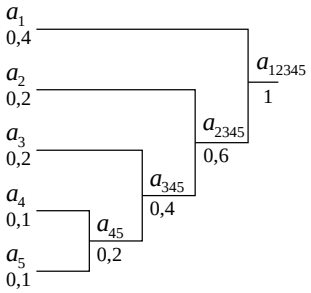
$$a_1 \equiv 0,4 \qquad a_2 \equiv 0,2 \qquad a_3 \equiv 0,2 \qquad a_4 \equiv 0,1 \qquad a_5 \equiv 0,1.$$

L'algorithme de Huffman effectue alors les opérations suivantes :

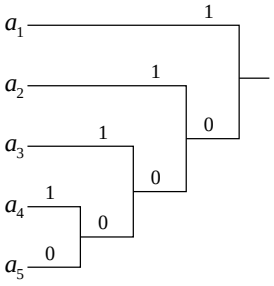
1. a_4 est combiné avec a_5 car ce sont les caractères ayant les plus petites fréquences d'apparition. Leur combinaison forme alors un nouveau symbole que nous appellerons a_{45} et dont la fréquence d'apparition dans le texte est $0,1 + 0,1 = 0,2$. On créé un arbre binaire ayant pour racine a_{45} et comme feuilles a_4 et a_5.
2. On a maintenant 3 symboles avec les probabilités 0,2 : a_2, a_3 et a_{45}. On doit en combiner deux, peu importe lesquels[1]. On va sélectionner arbitrairement les symboles a_3 et a_{45}. Ceux-ci en forment donc un nouveau, que nous appellerons a_{345} et dont la fréquence d'apparition est $0,2 + 0,2 = 0,4$.

[1] En fait, l'ordre peut être important pour certaines applications, comme nous le verrons plus loin. Cela dit, quels que soient les deux symboles que l'on combine, la taille du flux de sortie sera toujours la même.

3. On a maintenant deux symboles avec des fréquences de 0,4 : a_1 et a_{345}, plus un symbole, a_2, avec une fréquence de 0,2. On doit donc combiner a_2 avec, au choix, a_1 ou a_{345}. Choisissons de combiner a_2 et a_{345}. On obtient le symbole a_{2345} dont la fréquence est 0,6.
4. Enfin, il ne reste plus que deux symboles, a_1 et a_{2345}. On les combine pour obtenir a_{12345} dont la fréquence d'apparition est 1. À cette étape, l'arbre déterminant les codes des symboles est construit : les symboles d'origine sont les feuilles et chaque «combinaison» représente un noeud de l'arbre :



les symboles et leur fréquence



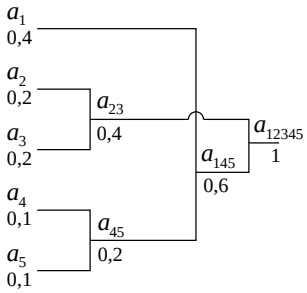
les bits associés à chaque arc

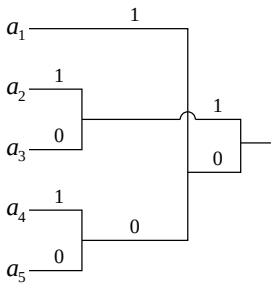
À chaque noeud de l'arbre, on code l'une des deux arêtes sortantes (une des arêtes sur la figure à droite ci-dessus) avec un 1 et l'autre avec un 0.

5. Pour obtenir le codage d'un symbole, il suffit de partir du noeud-symbole ayant une fréquence de 1 (autrement dit, la racine de l'arbre), et de parcourir l'arbre jusqu'à ce qu'on atteigne le symbole désiré. Au fur et à mesure du cheminement, on note les 0/1 des arcs parcourus. Ceux-ci forment le code correspondant au symbole. Ainsi, le code de a_3 sera obtenu en partant de a_{12345}, en allant sur la branche vers a_{2345} (on obtient le premier bit du code : 0), puis en allant vers a_{345} (bit 0), puis en allant vers a_3 (bit 1). Le code de a_3 sera donc 001. Les arbres ci-dessus nous donnent donc les codes suivants :

$$a_1 \equiv 1 \qquad a_2 \equiv 01 \qquad a_3 \equiv 001 \qquad a_4 \equiv 0001 \qquad a_5 \equiv 0000.$$

Notons que l'on aurait pu obtenir d'autres codes si l'on avait fait d'autres choix de combinaisons. Par exemple, si l'on avait combiné a_4 et a_5, puis a_2 et a_3, puis a_1 et a_{45}, puis a_{145} et a_{23}, on aurait obtenu l'arbre suivant :





On obtiendrait alors le codage suivant :

$$a_1 \equiv 01 \qquad a_2 \equiv 11 \qquad a_3 \equiv 10 \qquad a_4 \equiv 001 \qquad a_5 \equiv 000.$$

Quoi qu'il en soit, les fichiers de sortie auront la même taille. Pour montrer cela, il suffit de calculer l'espérance de gain (en bits/octet) des deux codages :

$$E(\text{code 1}) = 0,4 \times 1 + 0,2 \times 2 + 0,2 \times 3 + 0,1 \times 4 + 0,1 \times 4 = 2,2,$$
$$E(\text{code 2}) = 0,4 \times 2 + 0,2 \times 2 + 0,2 \times 2 + 0,1 \times 3 + 0,1 \times 3 = 2,2.$$

En moyenne, un octet du fichier de départ sera donc codé en sortie sur 2,2 bits. En fait, cette égalité des taux de compression n'est pas surprenante puisque ce qui importe dans l'algorithme, ce n'est pas vraiment le symbole qu'on code, mais sa fréquence d'apparition. ♦

2.2.1 Choix du codage de Huffman

Nous venons de voir que plusieurs codages de Huffman pouvaient exister pour compresser un même texte. La question qui vient alors immédiatement à toutes les lèvres est donc : «oui, certes, mais quel code est le meilleur ?». D'un point de vue taux de compression, comme nous venons de le voir sur l'exemple ci-dessus, tous les codes sont égaux. Par contre, lorsque l'on veut programmer Huffman, on s'aperçoit que :

Propriété : *Pour un ensemble de symboles et de fréquences donné, tous les codes de Huffman ont le même taux de compression. Cependant, le meilleur est en général celui dont la variance est la plus faible.*

Exemple 3 (suite) : Rappelons que la variance est la moyenne des carrés des différences à la moyenne. Par conséquent,

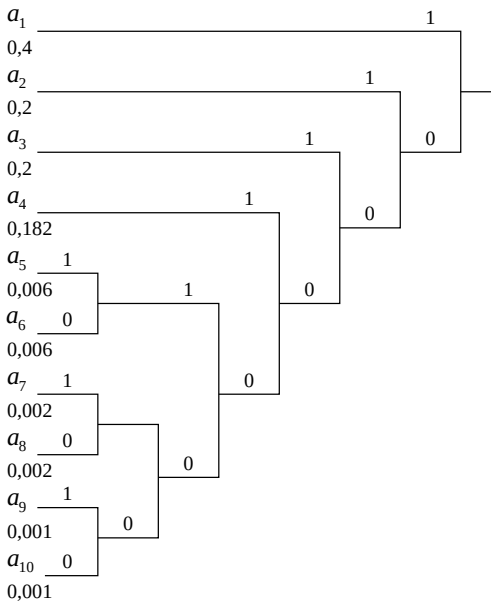
$$\begin{aligned} V(\text{code 1}) &= 0,4 \times (1-2,2)^2 + 0,2 \times (2-2,2)^2 + 0,2 \times (3-2,2)^2 \\ &\quad + 0,1 \times (4-2,2)^2 + 0,1 \times (4-2,2)^2 = 1,36, \\ V(\text{code 2}) &= 0,4 \times (2-2,2)^2 + 0,2 \times (2-2,2)^2 + 0,2 \times (2-2,2)^2 \\ &\quad + 0,1 \times (3-2,2)^2 + 0,1 \times (3-2,2)^2 = 0,16. \end{aligned}$$

Autrement dit, le meilleur code est le deuxième. ♦

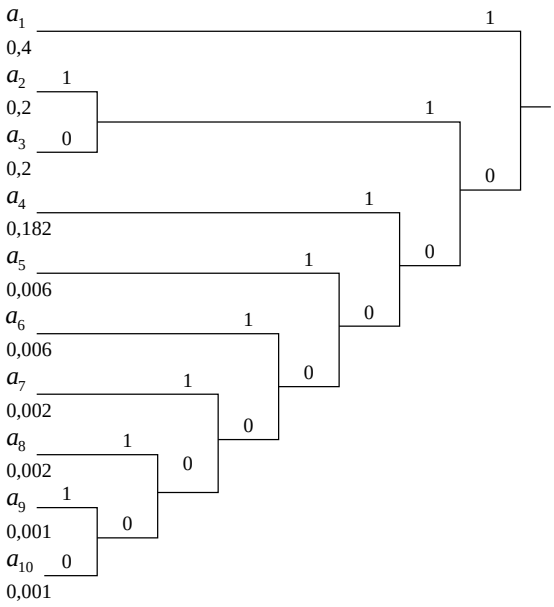
Pourquoi faut-il essayer de réduire au maximum la variance, me direz-vous ? Eh bien imaginons que nous ayons à programmer un logiciel de compression utilisant Huffman. Sur son flux d'entrée notre merveilleux logiciel reçoit un symbole, disons a_4. Supposons que l'arbre de Huffman correspondant au flux d'entrée soit celui en bas de la page précédente. Notre logiciel doit donc envoyer sur son flux de sortie le code «001». Oui, mais bon, quel algorithme doit-on utiliser pour obtenir ce code ? On pourrait partir de la racine de l'arbre, c'est-à-dire le noeud le plus à droite sur le dessin, et rechercher dans l'arbre le symbole en question. Clairement, effectuer une recherche du symbole dans l'arbre (avec la possibilité de parcourir tout l'arbre avant d'obtenir le code cherché) est totalement inefficace en terme de temps de réponse, et doit donc être proscrite. Un algorithme beaucoup plus intelligent, et c'est d'ailleurs celui qui est utilisé par tous les logiciels, est de partir du symbole reçu sur le flux d'entrée et de remonter l'arbre jusqu'à ce qu'on atteigne la racine. Comme il n'y a qu'un seul chemin entre le symbole et la racine, le temps de réponse de l'algorithme est excellent. Partons donc du symbole a_4 et remontons vers la racine. Sur le chemin, les bits rencontrés sont successivement «1», «0» et enfin «0», ce qui forme le code «100». Or, nous, ce que l'on voulait, c'était le code «001». Pas de problème me direz-vous, il suffit d'inverser l'ordre des bits rencontrés. Tout à fait exact, mais, pour pouvoir faire cela, il faut d'abord avoir stocké tous les bits. Par conséquent, il faut avoir un buffer dont la taille est au moins égale à la longueur du plus grand code de l'arbre. Et c'est là qu'intervient la variance : en général, plus la variance est faible, plus les tailles des différents codes sont proches les unes des autres (cf. le théorème de Bienaymé-Tchebicheff). La conséquence immédiate est, qu'en général, plus la variance est faible, plus la taille du buffer à allouer est petite. Ce fait est bien confirmé avec les codes 1 et 2 ci-dessus.

Attention : ce n'est qu'en général seulement que cette propriété est vérifiée. On peut trouver des exemples dans lesquels la propriété n'est pas vérifiée :

Exemple 4 : Considérons un alphabet de 10 symboles : $\{a_1, a_2, \ldots, a_{10}\}$ ainsi que les deux codes décrits sur la figure ci-après.



code 1



code 2

Alors, les gains des deux codes sont égaux à :

$$\begin{aligned} E(\text{code 1}) &= 0,4 + 0,2 \times 2 + 0,2 \times 3 + 0,182 \times 4 + 0,006 \times 6 + 0,006 \times 6 + 0,002 \times 7 \\ &\quad + 0,002 \times 7 + 0,001 \times 7 + 0,001 \times 7 = 2,242, \\ E(\text{code 2}) &= 0,4 + 0,2 \times 3 + 0,2 \times 3 + 0,182 \times 3 + 0,006 \times 4 + 0,006 \times 5 + 0,002 \times 6 \\ &\quad + 0,002 \times 7 + 0,001 \times 8 + 0,001 \times 8 = 2,242, \end{aligned}$$

et les variances sont égales à :

$$\begin{aligned} V(\text{code 1}) &= 0,4 \times (1-2,242)^2 + 0,2 \times (2-2,242)^2 + 0,2 \times (3-2,242)^2 + 0,182 \times (4-2,242)^2 \\ &\quad + 0,006 \times (6-2,242)^2 + 0,006 \times (6-2,242)^2 + 0,002 \times (7-2,242)^2 \\ &\quad + 0,002 \times (7-2,242)^2 + 0,001 \times (7-2,242)^2 + 0,001 \times (7-2,242)^2 = 1,605, \\ V(\text{code 2}) &= 0,4 \times (1-2,242)^2 + 0,2 \times (3-2,242)^2 + 0,2 \times (3-2,242)^2 + 0,182 \times (3-2,242)^2 \\ &\quad + 0,006 \times (4-2,242)^2 + 0,006 \times (5-2,242)^2 + 0,002 \times (6-2,242)^2 \\ &\quad + 0,002 \times (7-2,242)^2 + 0,001 \times (8-2,242)^2 + 0,001 \times (8-2,242)^2 = 1,150. \end{aligned}$$

Autrement dit, le deuxième code a une variance inférieure au premier. Cependant, le buffer utilisé pour coder les symboles doit être manifestement plus grand que celui utilisé dans le premier code. ♦

Un autre argument, peut-être plus convaincant, pour choisir le code ayant la plus petite variance est que certains logiciels de compression doivent transmettre sur des lignes de communication les données compressées au moment où elles sont générées (par exemple MNP7, un alogrithme de compression utilisé dans les transferts modem). Dans de telles situations, si le codage de Huffman a une grosse variance, l'encodeur génère des bits sur son flot de sortie avec des vitesses très fluctuantes. Comme, en principe, le taux de transfert sur la ligne de communication est constant, l'algorithme de compression doit utiliser un «gros» cache pour assurer qu'il y aura toujours des bits à transmettre sur la ligne de communication. Il est relativement aisé de voir «intuitivement» pourquoi cette assertion est vraie : supposons que la variance soit nulle. Rappelons que la variance est égale à :

$$V = \sum_{i=1}^{n} P(a_i) \times (\text{longueur du code de } a_i - E)^2, \qquad \text{où } E = \sum_{i=1}^{n} P(a_i) \times \text{longueur du code de } a_i.$$

Alors, les longueurs de tous les codes sont égales à E. Autrement dit, la longueur du cache n'a pas besoin d'être supérieure à E. Par contre, plus la variance est grande, plus les longueurs des codes de certains symboles

sont éloignés de la longueur moyenne. Comme la génération d'un code d'un symbole requiert de parcourir tout l'arbre de Huffman du symbole vers la racine de l'arbre, puis d'inverser l'ordre des bits, plus le nombre de bits du code est grand, plus sa génération est longue. Des longueurs de codes très fluctuantes entrainent donc que les insertions des codes dans le cache sont effectuées avec des vitesses elles-aussi très fluctuantes. Conséquence : le cache doit être de taille importante pour assurer qu'il y aura toujours des bits à transmettre à la ligne de communication.

2.2.2 Optimalité du codage de Huffman

Huffman n'est pas le meilleur algorithme de compression possible (par exemple les méthodes arithmétiques sont meilleures) car il code toujours les symboles sur un nombre entier de bits. Il suffit de lire l'algorithme pour s'en convaincre. Or, l'entropie étant calculée à partir de logarithmes, celle-ci a peu de chances d'être un nombre entier. Conclusion : un codage optimal devrait coder les symboles sur un nombre fractionnaire de bits. A priori, la chose paraît impossible mais, comme dirait Brassens, la suite vous prouvera que non. Bon, mais alors, si on peut faire mieux, pourquoi étudier Huffman, me direz-vous ? Eh bien, tout d'abord, c'est un algorithme «classique» et qui est utilisé dans beaucoup de logiciels. Ensuite, il est simple, rapide et tout de même assez efficace puisque, comme le montre le tableau ci-dessous, le taux de compression de l'ensemble du cours de probabilité obtenu grâce au programme de la sous-section 2.7.2 est d'environ 35% :

nom du fichier	taille du fichier d'origine		taille du fichier compressé par Huffman	
deust.tex	3809	octets	3178	octets
section1.tex	38142	octets	24898	octets
section2.tex	33569	octets	22403	octets
section3.tex	63915	octets	40914	octets
section4.tex	68099	octets	43991	octets
section5.tex	27463	octets	17800	octets
section6.tex	68757	octets	44469	octets
section7.tex	23096	octets	15277	octets
section8.tex	43803	octets	28779	octets
section9.tex	41920	octets	27149	octets
section10.tex	47436	octets	31635	octets
section11.tex	13442	octets	8157	octets
total	473451	octets	308650	octets

TAB. 9: Compression du cours de probabilité par l'algorithme de Huffman

Enfin, dans certains cas, Huffman est optimal :

Propriété : *Lorsqu'il y a au moins deux symboles et que les fréquences d'apparition de ceux-ci sont des puissances (négatives) de 2, l'algorithme de Huffman effectue une compression optimale.*

Démonstration : Prenons un texte sur un alphabet de n symboles $\{a_1, \ldots, a_n\}$ (on suppose dans la suite que tous les symboles sont utilisés, c'est-à-dire qu'ils ont des fréquences non nulles). Si l'alphabet n'a qu'un symbole, le résultat est évident. Supposons que toutes les fréquences, ou probabilités $P(a_i)$, soient des puissances de 2. Puisque l'alphabet est fini, il existe $i_0 \in \{1, \ldots, n\}$ tel que $P(a_{i_0}) \leq P(a_i)\ \forall i \neq i_0$. Supposons que cette inégalité soit stricte. Dans ce cas, $P(a_i) \geq 2 \times P(a_{i_0})\ \forall i \neq i_0$, ou encore :

$$\frac{P(a_i)}{P(a_{i_0})} \text{ est un nombre pair pour tout } i \neq i_0.$$

Or on sait que $\sum_{i=1}^{n} P(a_i) = P(a_{i_0}) + \sum_{i \neq i_0} P(a_i) = 1$. Donc :

$$\frac{P(a_{i_0}) + \sum_{i \neq i_0} P(a_i)}{P(a_{i_0})} = 1 + \sum_{i \neq i_0} \frac{P(a_i)}{P(a_{i_0})} = \frac{1}{P(a_{i_0})}.$$

Or, comme il y a au moins deux symboles ayant des fréquences non nulles, $P(a_{i_0}) < 1$, donc $\frac{1}{P(a_{i_0})}$ est un nombre pair (car $P(a_{i_0})$ est une puissance de 2). De même les $\frac{P(a_i)}{P(a_{i_0})}$ sont aussi des nombres pairs. Donc l'égalité ci-dessus est impossible. Par conséquent, notre assertion de départ, à savoir $P(a_i) > P(a_{i_0})\ \forall i \neq i_0$ est fausse. Autrement dit, il existe un $i_1 \neq i_0$ tel que $P(a_{i_1}) = P(a_{i_0})$.

Puisque les symboles a_{i_0} et a_{i_1} ont les fréquences les plus petites, Huffman peut les combiner, pour former un symbole $a_{i_{01}}$ ayant pour fréquence $2 \times P(a_{i_0})$, autrement dit encore une puissance de 2. On peut alors recommencer récursivement le processus ci-dessus avec l'alphabet $\{a_1, \ldots, a_n, a_{i_{01}}\} \backslash \{a_{i_0}, a_{i_1}\}$.

Montrons maintenant que le nombre de bits k_i utilisé pour coder les a_i est exactement égal à $-\log_2 P(a_i)$. Pour cela, il nous faut parcourir l'arbre de Huffman construit dans les paragraphes précédents. Appelons $\{b_1, \ldots, b_p\}$ l'ensemble des symboles contenus dans tous les noeuds de l'arbre, c'est-à-dire les a_i, mais aussi les $a_{i_{01}}$ et ainsi de suite. Partons de la racine de l'arbre. D'après ce qui précède, les deux symboles «en dessous» de cette racine ont tous les deux pour fréquences 1/2 et ont un code de 1 bit. Donc la propriété $k_i = -\log_2 P(b_i)$ est vérifiée pour ces deux symboles. Lorsqu'un noeud/symbole b_i a au moins un fils, on est sûr d'après ce qui précède qu'il a exactement deux fils, appelons les b_{i_1} et b_{i_2}, et que les 2 fils ont la même fréquence, à savoir $P(b_i)/2$. De plus, ces deux symboles sont codés sur $k_i + 1$ bits. On a bien $k_{i_1} = k_{i_2} = 1 - \log_2 P(b_i) = -\log_2(P(b_i)/2)$. Donc par récurrence, pour tout $i \in \{1, \ldots, p\}$, $k_i = -\log_2 P(b_i)$. Or comme $\{a_i, \ldots, a_n\} \subset \{b_1, \ldots, b_p\}$, la propriété reste vraie pour tous les a_i.

Il ne nous reste plus maintenant qu'à calculer la redondance du codage de Huffman et à montrer que celle-ci est égale à 0. Ainsi, on aura montré que ce codage est optimal :

$$R = \sum_{i=1}^{n} P(a_i)k_i - \sum_{i=1}^{n} (-P(a_i)\log_2 P(a_i)) = \sum_{i=1}^{n} (-P(a_i)\log_2 P(a_i)) - \sum_{i=1}^{n} (-P(a_i)\log_2 P(a_i)) = 0.$$

♦

D'une manière générale, l'algorithme de Huffman n'est pas optimal, c'est-à-dire qu'il existe des algorithmes permettant d'obtenir de meilleurs taux de compression. Cependant, il est en géréral très proche de l'optimum. De plus, il est l'optimum d'une classe particulière de compresseurs :

Propriété : *Soient $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ un alphabet à n symboles et $\mathcal{M}$ un message de taille finie r sur cet alphabet contenant chacun des symboles de $\mathcal{A}$. Soit $\mathcal{C}$ la classe des algorithmes de compression sans perte de données codant chaque symbole a_i par un autre symbole b_i. L'algorithme de Huffman est un optimum de $\mathcal{C}$ pour le message $\mathcal{M}$, c'est-à-dire qu'aucun autre algorithme de $\mathcal{C}$ ne permet d'obtenir un taux de compression de $\mathcal{M}$ strictement inférieur à celui de Huffman.*

Démonstration : La démonstration ci-dessous est adaptée du polycopié de DEA de Jean-Yves Jaffray : «Introduction à la théorie de l'information». Elle est effectuée en trois parties : dans la première, on montre qu'il existe dans $\mathcal{C}$ un algorithme optimal. Dans la deuxième, on montre qu'un code optimal peut être décrit, comme Huffman, à l'aide d'un arbre binaire dont les feuilles représentent les symboles de $\mathcal{A}$. Enfin, dans la troisième partie, on montre que tout arbre optimal peut être transformé en code de Huffman sans augmenter le taux de compression.

Première partie : il existe un algorithme optimal dans $\mathcal{C}$

Appelons P_i les fréquences d'apparition des symboles a_i dans le texte. Un algorithme C de $\mathcal{C}$ transformant les a_i en b_i est «meilleur» qu'un autre code C' de $\mathcal{C}$ les transformant en b'_i si et seulement si il fournit en sortie un texte de longueur inférieure au deuxième, autrement dit si et seulement si :

$$\sum_{i=1}^{n} P_i|b_i| \times |\mathcal{M}| \leq \sum_{i=1}^{n} P_i|b'_i| \times |\mathcal{M}|.$$

Dans Huffman, les b_i sont de tailles finies, donc la quantité $T = \sum_{i=1}^{n} P_i|b_i|$ fournie par Huffman est finie. Puisque chaque P_i est strictement positif, un algorithme C' transformant les a_i en b'_i et meilleur qu'Huffman vérifie :

$$P_k|b'_k| \leq \sum_{i=1}^{n} P_i|b'_i| \leq T,\ \forall\ k \in \{1, \ldots, n\}.$$

Par conséquent, il existe un nombre fini de b_k possibles et donc un nombre fini d'algorithmes de $\mathcal{C}$ «meilleurs» que Huffman. Conséquence : il existe un algorithme optimal dans $\mathcal{C}$.

Deuxième partie : optimum ⇒ arbre binaire

Soit un code C de $\mathcal{C}$ optimal ne pouvant être décrit par un arbre binaire. Ce code associe à chaque a_i un symbole b_i de k_i bits. On sait que $k_i \geq 1$ sinon le code ne serait pas inversible et il pourrait y avoir perte de données. On sait en outre que les k_i sont finis d'après la partie précédente.

Maintenant, tous les b_i sont de taille finie supérieure ou égale à 1. On peut donc effectuer une partition de l'ensemble des a_i en deux sous-ensembles : le premier, $\mathcal{A}_1^0$, regroupe les a_i dont les b_i correspondant commencent par un bit à 0, le deuxième, noté $\mathcal{A}_1^1$, ceux dont les b_i commencent par un bit à 1. Si les deux groupes sont non vides, on peut donc représenter le code C grâce à un arbre dont la racine symbolise $\mathcal{A}$ et dont les sous-arbres gauche et droit représentent respectivement $\mathcal{A}_1^0$ et $\mathcal{A}_1^1$. On peut alors recommencer le processus avec $\mathcal{A}_1^0$ et $\mathcal{A}_1^1$ en se fondant sur le deuxième bit des b_i. Si, à chaque étape, soit $\mathcal{A}_k^0$ (resp. $\mathcal{A}_k^1$) peut être partitionné en deux sous-ensembles, soit $\mathcal{A}_k^0$ (resp. $\mathcal{A}_k^1$) est de cardinal 1, alors on obtient un arbre binaire comme indiqué au début de la démonstration.

Supposons maintenant qu'un des $\mathcal{A}_k^0$ (la démonstration est identique pour $\mathcal{A}_k^1$) soit vide. Cela veut dire que tous les symboles b_i des a_i appartenant à $\mathcal{A}_k$ ont leur $k^{\text{ème}}$ bit à 1. Deux cas peuvent se présenter : soit tous les b_i sont de tailles strictement supérieures à k. Dans ce cas, le code n'est pas optimal car cela veut dire que le $k^{\text{ème}}$ bit de chaque b_i n'est pas discriminant et qu'un code C' obtenu à partir de C en enlevant le $k^{\text{ème}}$ bit de ces b_i est toujours sans perte de données, inversible, et augmente strictement le taux de compression du message $\mathcal{M}$. Si, maintenant, certains des b_i sont de taille k, le code C' obtenu à partir de C en changeant le $k^{\text{ème}}$ bit de ces b_i en 0 est toujours sans perte de données, inversible, et vérifie bien la récurrence du paragraphe précédent. Conclusion : lorsqu'un code est optimal, il peut être décrit sous la forme d'un arbre binaire comme celui de l'algorithme de Huffman.

Troisième partie : Huffman est optimal

On sait déjà qu'il existe un algorithme optimal dans $\mathcal{C}$ et que celui-ci peut être décrit par un arbre binaire. Appelons rang d'un sommet la longueur du chemin reliant le sommet à la racine de l'arbre. Si l'on classe les symboles a_i par ordre de fréquence (probabilité) croissante (les ex-aequo sont départagés arbitrairement), alors on appellera les k symboles les moins probables les k premiers symboles dans le classement.

Nous allons montrer maintenant qu'il existe un algorithme optimal dans lequel les 2 symboles les moins probables font partie des sommets de rang maximal, R, de l'arbre. Soit donc un algorithme optimal de $\mathcal{C}$ associant aux a_i des b_i et d'arbre binaire de rang maximal R. Comme l'arbre est binaire, il existe au moins deux sommets de rang R. Supposons maintenant qu'il existe un a_i, l'un des 2 symboles les moins probables, de rang $r < R$ (autrement dit, $|b_i| = r$). Puisqu'il existe deux sommets de rang R, cela implique qu'il existe a_j de rang R ne faisant pas partie des 2 symboles les moins probables. Autrement dit, $P_j \geq P_i$. Intervertissons les deux symboles dans l'arbre binaire (et modifions les sommets-ensembles de symboles en amont en conséquence). Ce nouveau code transforme $\mathcal{M}$ en un texte de longueur :

$$\begin{aligned} L' &= \left[P_i R + P_j r + \textstyle\sum_{k \neq i,j} P_k |b_k|\right] |\mathcal{M}| = \left[P_i R + P_j r - P_i r - P_j R + \textstyle\sum_{k=1}^{n} P_k |b_k|\right] |\mathcal{M}| \\ &= [(P_j - P_i)(r - R)]|\mathcal{M}| + L \leq L, \end{aligned}$$

où L était la longueur du texte issu de $\mathcal{M}$ après utilisation du code de départ. Par conséquent, soit l'inégalité ci-dessus est stricte et le code de départ ne pouvait être optimal, soit il y a égalité entre L et L' et le nouveau code est lui aussi optimal. On peut alors réitérer les échanges jusqu'à ce que la propriété annoncée soit vérifiée.

D'après l'expression de L (ou de L'), une permutation des sommets de même rang (accompagnée des modifications adéquates en amont) ne change pas la longueur du texte en sortie. Par conséquent, il existe parmi les algorithmes optimaux de $\mathcal{C}$ un algorithme pour lequel les 2 symboles a_i les moins probables ont rang R et un même antécédant dans l'arbre : leur union a_{ij}. Montrons maintenant que tout algorithme de $\mathcal{C}$ composé :

- de deux arcs d'extrémités les deux symboles a_i et a_j les moins probables et d'origine leur union a_{ij}, d'une part,
- d'un arbre binaire dont les feuilles sont le noeud a_{ij} ainsi que les $n-2$ autres symboles de $\mathcal{A}$ d'autre part,

est optimal. Soit un algorithme C optimal pour lequel les 2 symboles a_i et a_j les moins probables ont rang R et leur union a_{ij} comme antécédant. Si l'on ampute de l'arbre les deux arcs sortant de l'union a_{ij}, alors l'algorithme reste encore optimal si l'on considère que l'alphabet est $\mathcal{A}' = \{a_k, k \neq i, j\} \cup \{a_{ij}\}$ et que les symboles a_i et a_j de $\mathcal{M}$ sont transformés en a_{ij}. En effet, La longueur L du texte issu de $\mathcal{M}$ avant amputation et la longueur L' après amputation vérifient :

$$L = L' + [RP_i + RP_j - (R-1)P(a_{ij})]|\mathcal{M}| = L' + [RP_i + RP_j - (R-1)(P_i + P_j)]|\mathcal{M}| = L' + (P_i + P_j)|\mathcal{M}|.$$

Or, tout algorithme pour l'alphabet $\mathcal{A}'$ peut être complété par deux arcs sortant de a_{ij} et fournir un algorithme pour l'alphabet $\mathcal{A}$, de longueur la sienne plus $P(a_{ij})$. Donc si L est minimum, L' l'est aussi et un algorithme optimal pour $\mathcal{A}'$ complété de deux arcs sortant de a_{ij} est optimal pour $\mathcal{A}$. En réduisant récursivement $\mathcal{A}$ comme on vient de le voir, on montre donc l'optimalité de l'algorithme de Huffman. ♦

2.3 Le codage de Huffman adaptatif

Comme nous l'avions évoqué dans la section 1, la durée de compression est un facteur à ne pas négliger. Or, dans l'algorithme de Huffman tel que décrit précédemment, celle-ci peut être relativement conséquente car on doit effectuer deux passes sur le flux d'entrée :

- on lit une première fois l'ensemble du flux d'entrée afin de déterminer les fréquences d'apparition de chacun des symboles ;
- on lit une deuxième fois le flux d'entrée et, là, grâce aux fréquences collectées à l'étape précédente, on écrit le flux de sortie.

Lorsque l'on compresse des «petits» fichiers, ces deux étapes donnent des durées de compression relativement acceptables. Par contre, ce n'est plus le cas lorsque les fichiers du flux d'entrée sont grands. Par exemple, sur une RedHat 6.0 avec un processeur Pentim II 233, la lecture/ecriture d'un fichier d'1 Mo (lecture + écriture par tranches de 1 Ko) prend environ 0,07 secondes. Lire deux fois le fichier n'est pas dérangeant pour l'utilisateur. Par contre, la lecture/écriture d'un fichier de 100 Mo prend environ 37 secondes et, là, on a intérêt à limiter au maximum les accès disques. C'est pourquoi, souvent, on n'utilise pas la méthode de Huffman décrite ci-dessus mais plutôt une variante, la *méthode de Huffman adaptative* qui, elle, ne nécessite qu'une seule lecture du flux d'entrée.

L'idée de la méthode de Huffman adaptative est que le compresseur et le décompresseur débutent avec un arbre de Huffman vide, et qu'ils modifient cet arbre au fur et à mesure que les symboles sont lus sur le flot d'entrée (cette modification est effectuée en fonction des fréquences des symboles lus depuis le début de l'algorithme). Pour que la méthode fonctionne, le compresseur et le décompresseur doivent être synchronisés dans la mesure où ils doivent modifier l'arbre de la même manière lorsqu'ils reçoivent les symboles sur leurs flots d'entrée respectifs.

L'algorithme du compresseur est le suivant :

1. On part d'un arbre de Huffman vide. Aucun symbole n'a de code (compressé).
2. Le premier symbole sur le flux d'entrée est écrit tel quel sur le flux de sortie. On l'ajoute alors dans l'arbre de Huffman et on lui affecte un code.
3. Chaque fois qu'on lit un symbole, on regarde s'il appartient à l'arbre de Huffman. S'il n'en fait pas partie, on l'écrit tel quel et on le rajoute avec un nouveau code dans l'arbre de Huffman, sinon on met à jour la fréquence d'apparition du symbole.
4. Si besoin est, on réarrange l'arbre de manière à ce que ce soit toujours un arbre de Huffman.
5. On revient à l'étape 3.

L'algorithme du décompresseur effectue exactement les mêmes opérations, dans le même ordre. Quand il rencontre un symbole non compressé, il l'ajoute à l'arbre et lui affecte un code. Lorsqu'il lit sur son flux d'entrée un symbole compressé, il met à jour sa fréquence et réarrange l'arbre.

Il subsiste tout de même un léger problème : comment reconnaître un code compressé d'un symbole non compressé ? Le plus simple, et c'est ce qu'utilise cette méthode, est de faire précéder les symboles non compressés d'un caractère d'échappement (le fameux @ de la section 1). Quand le décompresseur rencontre ce code, il sait alors que l'octet qui le suit est un caractère sur 8 bits. Bon, on a résolu un problème, mais en faisant cela on

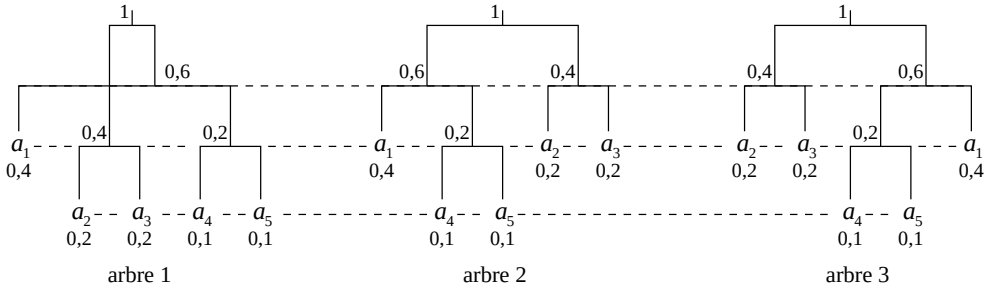
en a créé un autre : quel code faut-il donner au caractère d'échappement ? Il faut surtout éviter que ce symbole soit un des codes de l'arbre de Huffman. Or, comme l'arbre peut changer à chaque étape, il est difficile de trouver *a priori* un symbole qui marchera tout au long de l'algorithme. Afin de pallier cela, l'idée est d'ajouter le caractère d'échappement dans l'arbre de Huffman et de toujours lui affecter une fréquence d'apparition de 0. Conséquence : comme l'arbre de Huffman est potentiellement modifié à chaque étape de l'algorithme, le code utilisé pour le caractère d'échappement n'est pas fixe tout au long de l'algorithme.

Avant de traiter un exemple illustrant la méthode, nous allons voir le point délicat de l'algorithme, à savoir comment modifier l'arbre pour obtenir à coup sûr un arbre de Huffman.

2.3.1 Modification de l'arbre de Huffman

La principale différence entre les méthodes de Huffman non adaptatives et adaptatives est que, dans la deuxième, on vérifie à chaque nouveau symbole lu sur le flot d'entrée que l'arbre contenant les symboles et les codes est toujours un arbre de Huffman. Si ce n'est plus le cas, alors on doit le modifier pour que ce soit à nouveau un arbre de Huffman. Oui, mais pour cela, il faut caractériser précisément ce qu'est un arbre de Huffman.

Ci-dessous voici différentes représentations de l'arbre de la page 29 :



La première remarque que l'on peut faire est que si un arbre est de Huffman, alors sur chaque niveau de l'arborescence (les lignes en pointillés) les fréquences associées aux noeuds sont inférieures aux fréquences de tous les noeuds des niveaux supérieurs. C'est bien normal puisque ce que l'on veut faire avec Huffman, c'est associer les codes les plus petits aux symboles ayant le plus d'occurrences (les fréquences les plus élevées). Plus les fréquences sont élevées, plus on doit se trouver en haut de l'arbre. La deuxième remarque que l'on peut faire est que la fréquence associée à chaque noeud qui n'est pas une feuille est égale à la somme des fréquences de ses deux fils. Ces deux propriétés caractérisent totalement les arbres de Huffman :

Propriété : *Un arbre est dit de Huffman s'il vérifie les deux propriétés suivantes :*

1. *les fréquences associées aux noeuds d'un niveau d'arborescence sont inférieures ou égales aux fréquences de tous les noeuds des niveaux supérieurs ;*
2. *la fréquence associée à chaque noeud qui n'est pas une feuille est égale à la somme des fréquences de ses deux fils.*

On peut donc maintenant envisager un algorithme permettant de transformer un arbre en arbre de Huffman. Malheureusement, un tel algorithme serait parfaitement inefficace car il devrait effectuer beaucoup trop de tests. En effet, voyons ce qui se passe sur la figure 5. Sur cette figure, les nombres entre parenthèses correspondent aux nombres d'occurrences des différents symboles déjà lus sur le flot d'entrée. Supposons que l'on ait déjà été amené à construire l'arbre le plus à gauche sur la figure. Sur le flot d'entrée, on lit un nouveau symbole a_1. Le nombre d'occurrences de ce symbole passe donc de 1 à 2. On remarque que sur le niveau d'arborescence au dessus du noeud a_1, tous les nombres d'occurrences sont supérieurs ou égaux à 2. On n'a donc pas *a priori* à modifier l'arbre. Mettons à jour le nombre d'occurrences du noeud parent de a_1 ; celui-ci passe à 3. Là encore, les niveaux d'arborescence au dessus ont plus d'occurrences, donc pas de modification à faire sur l'arbre. En continuant à remonter l'arbre, on s'aperçoit qu'il est bien de Huffman puisqu'il satisfait la propriété ci-dessus. On obtient l'arbre 2. On lit alors un nouveau symbole a_1 sur le flot d'entrée. Le nombre d'occurrences de a_1 passe alors à 3. Là, il y a des noeuds sur le niveau d'arborescence au dessus de a_1 qui ont moins de 3 occurrences. Il faut donc modifier l'arbre. Pour cela, le plus simple est d'intervertir a_1 avec l'un de ces noeuds (ici j'ai choisi

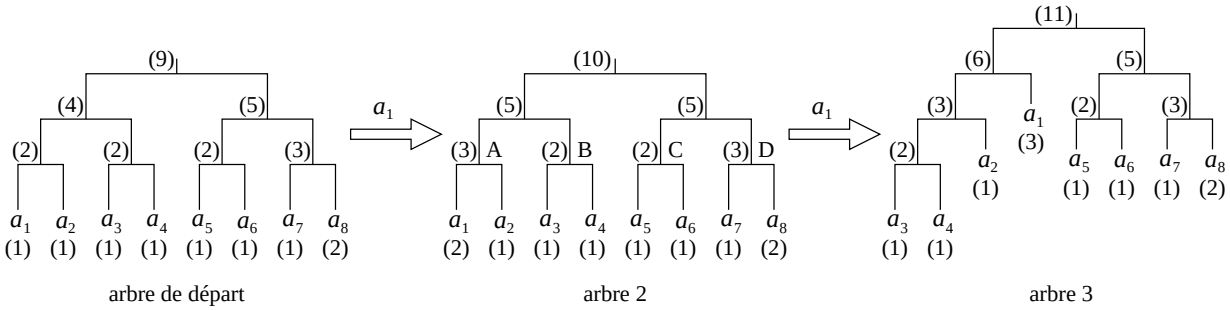


FIG. 5 – Tests à effectuer pour obtenir un arbre de Huffman.

le noeud parent de a_3 et a_4). Lorsque l'intervertion est faite, il faut encore s'assurer que les niveaux supérieurs ont tous des nombres d'occurrence supérieurs ou égaux à 3, puis recommencer le processus avec les ancêtres de a_1 jusqu'à ce qu'on arrive à la racine.

STOP ! Vous avez vu combien on a dû faire de tests ? Pour savoir si le nombre d'occurrences de a_1 est inférieur ou égal à ceux des niveaux supérieurs de l'arborescence, on a été obligé de balayer tous les noeuds du niveau d'arborescence juste au dessus de a_1. Pour le passage de l'arbre 2 à l'arbre 3, par exemple, on a été obligé d'examiner les noeuds A, B, C et D. C'est beaucoup trop long pour que l'algorithme soit efficace. En fait, on irait beaucoup plus vite si l'arbre était mieux trié : il est déjà trié «verticalement» puisque plus l'on descend dans l'arbre et plus le nombre d'occurrences diminue. On va rajouter maintenant un tri horizontal : sur un niveau d'arborescence donné, plus on va aller vers la droite, plus le nombre d'occurrences va être élevé. Dans ce cas, les tests se limitent à examiner les noeuds à droite sur le même niveau d'arborescence et, éventuellement, ceux les plus à gauche du niveau supérieur. Là on obtient un algorithme efficace.

Propriété : *Un arbre est dit de Huffman adaptatif s'il vérifie les trois propriétés suivantes :*

1. *les fréquences associées aux noeuds d'un niveau d'arborescence sont inférieures aux fréquences de tous les noeuds des niveaux supérieurs ;*
2. *la fréquence d'un noeud donné est supérieure ou égale à celle des noeuds qui sont à sa gauche sur le même niveau d'arborescence ;*
3. *la fréquence associée à chaque noeud qui n'est pas une feuille est égale à la somme des fréquences de ses deux fils.*

On a vu précédemment comment faire les modifications : par intervertion de noeuds. On peut donc maintenant donner l'algorithme de transformation de l'arbre :

Algorithme 1 (modification de l'arbre de Huffman) *À l'arrivée d'un symbole sur le flot d'entrée, appelons-le a, dont le nombre d'occurrences était auparavant n, on effectue les opérations suivantes :*

1. *On compare a avec ses successeurs immédiats (par rapport au nombre d'occurrences) dans l'arbre (de la gauche vers la droite et de bas en haut). Si le successeur a un nombre d'occurrences supérieur ou égal à $n+1$, alors les noeuds sont encore triés dans le bon ordre et il n'y a pas besoin de modifier l'arbre. Sinon certains successeurs ont un nombre d'occurrences égal à celui de a. Dans ce cas, on intervertit le dernier successeur (celui le plus haut, puis le plus à droite dans l'arbre) qui n'est pas un de ses ancêtres avec a.*
2. *On incrémente n.*
3. *Si a est la racine de l'arbre, l'algorithme de modification est terminé. Sinon, on repart à l'étape 1 mais avec pour nouveau a le parent de a.*

Examinons sur un exemple comment fonctionne cet algorithme : Partons de l'arbre le plus à gauche de la figure 6. Notons que cet arbre vérifie bien la propriété des arbres de Huffman adaptatifs. En effet, sur chaque niveau d'arborescence, les nombres d'occurrences sont triés par ordre croissant de la gauche vers la droite, et du bas vers le haut. On lit maintenant le symbole a_1 sur le flot d'entrée. Appelons n_i les nombres d'occurrence des a_i. Étape 1 de l'algorithme : comparons n_1 et n_2. $n_2 = 2 \geq n_1 + 1 = 1 + 1 = 2$. Donc l'arbre n'a pas

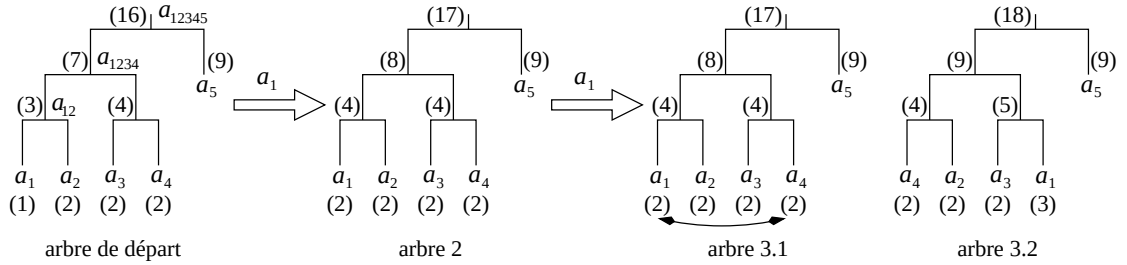


FIG. 6 – Illustration de l'algorithme de modification.

besoin d'être modifié. Étape 2 : on fait passer n_1 de 1 à 2. Étape 3 : a_1 n'est pas la racine de l'arbre. On recommence donc l'algorithme avec le noeud a_{12}. $n_{12}+1 = 3+1 = 4 \leq n_{34}$. Donc pas de modification de l'arbre. Étape 2 : on incrémente n_{12}. Étape 3 : a_{12} n'est pas la racine, donc on recommence l'étape 1 avec le noeud a_{1234} : $n_{1234}+1 = 7+1 = 8 \leq n_5 = 9$. Donc pas de modification de l'arbre. Étape 2 : on incrémente n_{1234}. Étape 3 : a_{1234} n'est pas la racine, donc on recommence avec a_{12345}. Ce noeud n'a pas de successeur, donc on passe à l'étape 2. on incrémente n_{12345}. Étape 3 : on est à la racine, donc l'algorithme est terminé. On obtient alors l'arbre 2 sur la figure 6.

On lit maintenant un nouveau symbole a_1 sur le flot d'entrée. Étape 1 : on examine les successeurs de a_1. $n_2 = 3 < n_1+1 = 4$, donc l'arbre doit être modifié : on doit intervertir a_1 avec le noeud ayant exactement n_2 occurrences le plus haut et le plus à droite dans l'arbre. a_2 est un successeur ; on continue vers la droite : a_3 est un successeur ; on continue vers la droite : a_4 est un successeur ; on continue avec le noeud le plus à gauche sur le niveau d'arborescence supérieur : a_{12} n'est pas un successeur car $n_{12} = 4 \geq n_1+1 = 2+1$. Donc on échange a_1 avec a_4. Étape 2 : on incrémente n_1. Étape 3 : a_1 n'est pas la racine de l'arborescence, donc on recommence l'étape 1 avec le parent de a_1, à savoir a_{13}. Et ainsi de suite. On obtient alors l'arbre 3.2.

Sur la figure ci-dessous, on continue à lire sur le flot d'entrée le symbole a_1. Étape 1 : on examine les

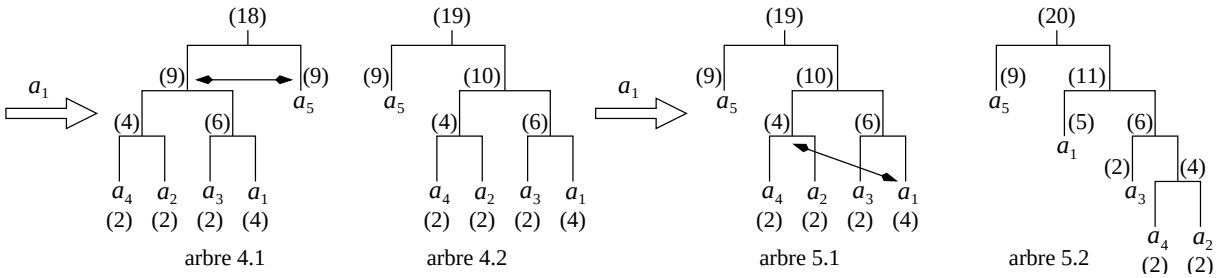


FIG. 7 – Illustration de l'algorithme de modification (suite).

successeurs de a_1, c'est-à-dire a_{42}. $n_{42} = 4 \geq n_1+1 = 3+1 = 4$. Donc pas de modification de l'arbre. Étape 2 : on incrémente n_1, qui passe donc de 3 à 4. Étape 3 : a_1 n'est pas la racine de l'arborescence, donc on recommence l'étape 1 avec le noeud a_{13}. Étape 1 : on compare a_{13} avec son successeur immédiat, c'est-à-dire a_{1234}. $n_{1234} \geq n_{13}+1$. Donc pas de modification de l'arbre. Étape 2 : on incrémente n_{13}, qui passe à 6. On obtient alors l'arbre 4.1. Étape 3 : a_{13} n'est pas la racine, donc on recommence avec a_{1234}. $n_{1234}+1 = 9+1 > n_5$. a_5 est l'unique successeur de a_{1234}. Donc on intervertit a_{1234} et a_5. Étape 2 : on incrémente n_{1234}, qui passe donc à 10. Étape 3 : a_{1234} n'est pas la racine, donc on recommence avec a_{12345}. Ce dernier n'a pas de successeur, donc on passe directement à l'étape 2 : on incrémente n_{12345} et l'algorithme est fini. On obtient alors l'arbre 4.2.

Pour terminer, lisons encore une dernière fois le symbole a_1. Comparons a_1 avec ses successeurs immédiats, à savoir a_{42}. $n_{42} = 4 < n_1+1 = 4+1 = 5$. Donc on intervertit a_{42} et a_1. Étape 2 : on incrémente n_1, qui passe donc à 5. Étape 3 : a_1 n'est pas la racine, donc on recommence l'étape 1 avec le parent de a_1, à savoir a_{1234}. On compare a_{1234} avec son successeur immédiat, à savoir a_{12345}. $n_{12345} = 19 \geq n_{1234}+1 = 10+1$. Donc pas de modification de l'arbre. Étape 2 : on incrémente n_{1234}. Étape 3 : a_{1234} n'est pas la racine, donc on recommence avec son parent a_{12345}. Celui-ci n'a pas de successeur, donc on passe à l'étape 2. On incrémente n_{12345} et l'algorithme est terminé.

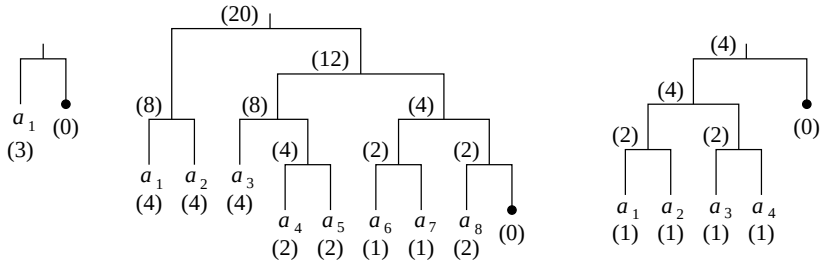
2.3.2 Insertion de nouveaux symboles

Nous avons vu dans la sous-section précédente comment modifier l'arbre lors de l'arrivée de symboles appartenant déjà à l'arbre de Huffman. Il nous reste encore à voir comment réaliser l'insertion de nouveaux symboles dans l'arbre de Huffman. Cela pose deux problèmes :

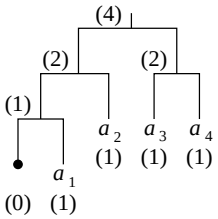
1. où placer le caractère d'échappement dans l'arbre ?
2. comment insérer les noeuds correspondant aux nouveaux symboles ?

On peut résoudre le premier problème de plusieurs manières. David Salomon, dans son livre «Data Compression : the complete reference» propose d'associer au caractère d'échappement un nombre d'occurrences toujours égal à 0 et de toujours le placer sur la branche de l'arbre ne contenant que des «0».

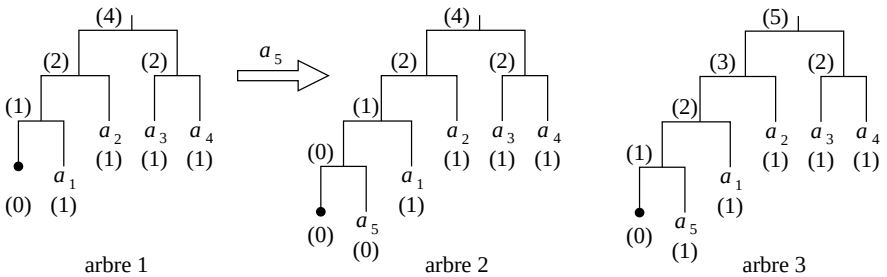
Si l'on affectait des «0» aux branches de droite et des «1» aux branches de gauche, on obtiendrait ainsi des arbres tels que ceux ci-dessous (le • représente le caractère d'échappement) :



Une autre manière de résoudre le problème, et c'est ce que nous utiliserons par la suite, est de considérer que le caractère d'échappement est un caractère comme un autre et de le placer dans l'arbre en fonction de son nombre d'occurrences (qui reste, rappelons-le toujours à 0). L'arbre le plus à droite sur la figure ci-dessus serait alors remplacé par :

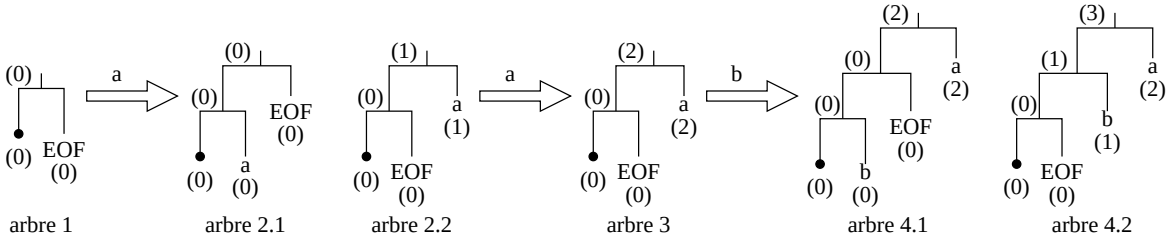


Voyons maintenant comment résoudre le deuxième problème, à savoir où insérer les nouveaux symboles dans l'arbre. On sait que si l'on considère que le caractère d'échappement est un symbole comme un autre, il est forcément au niveau le plus bas de l'arborescence (car sa fréquence est de 0 et que l'arbre vérifie la propriété d'arbre de Huffman adaptatif). Par conséquent, si l'on remplace ce symbole par un noeud ayant pour enfant d'une part le caractère d'échappement et, d'autre part, le nouveau symbole avec une fréquence de 0, l'arbre sera toujours de Huffman adaptatif et il suffira alors de lancer la procédure de modification de l'arbre pour le mettre à jour correctement. Exemple : on part de l'arbre 1 ci-dessous et on lit sur le flot d'entrée un nouveau symbole a_5. On remplace alors le caractère d'échappement par le noeud contenant le nouveau symbole et le caractère d'échappement, ce qui nous donne l'arbre 2. On effectue alors l'algorithme de modification de l'arbre, ce qui nous donne l'arbre 3.



2.3.3 Exemple d'application de la méthode

On veut compresser la chaîne de caractères «aabccbb». On devrait commencer l'algorithme avec un arbre vide ou, à la rigueur, avec un arbre ne contenant que le caractère d'échappement. Mais cela entraînerait, par la suite, de nombreux tests inutiles (tester si l'arbre est vide dans le premier cas, et s'il ne contient qu'une seule feuille dans le deuxième). Pour éviter cela, nous allons partir de l'arbre 1 ci-après ayant deux feuilles : le caractère d'échappement et le symbole de fin de fichier «EOF». On supposera par la suite que, dans l'arbre, toutes les branches de gauche sont codées avec des «0» et que toutes les branches de droite sont des «1».



On lit maintenant le caractère «a» sur le flot d'entrée. Ce symbole ne fait pas partie de l'arbre. On écrit donc sur le flot de sortie le caractère d'échappement puis le code ASCII correspondant au «a», autrement dit «0|01100001». On rajoute le caractère «a» dans l'arbre (cf. arbre 2.1) et on effectue la modification de l'arbre afin de se retrouver avec un arbre de Huffman adaptatif. On obtient alors l'arbre 2.2. On lit maintenant un nouveau caractère sur le flot d'entrée. C'est encore un «a». Cette fois-ci, le caractère appartient à l'arbre. On écrit donc son code sur le flot de sortie : «1» et on met à jour l'arbre, qui devient l'arbre 3. On lit maintenant un «b» sur le flot d'entrée. Ce caractère n'appartient pas à l'arbre. On écrit donc sur le flot de sortie le caractère d'échappement suivi du code ASCII de «b» : «00|01100010». Notons que le code du caractère d'échappement n'est pas le même que celui utilisé lors de l'arrivée du premier «a». On rajoute «b» dans l'arbre, ce qui nous donne l'arbre 4.1 et on applique l'algorithme de modification qui nous donne l'arbre 4.2.

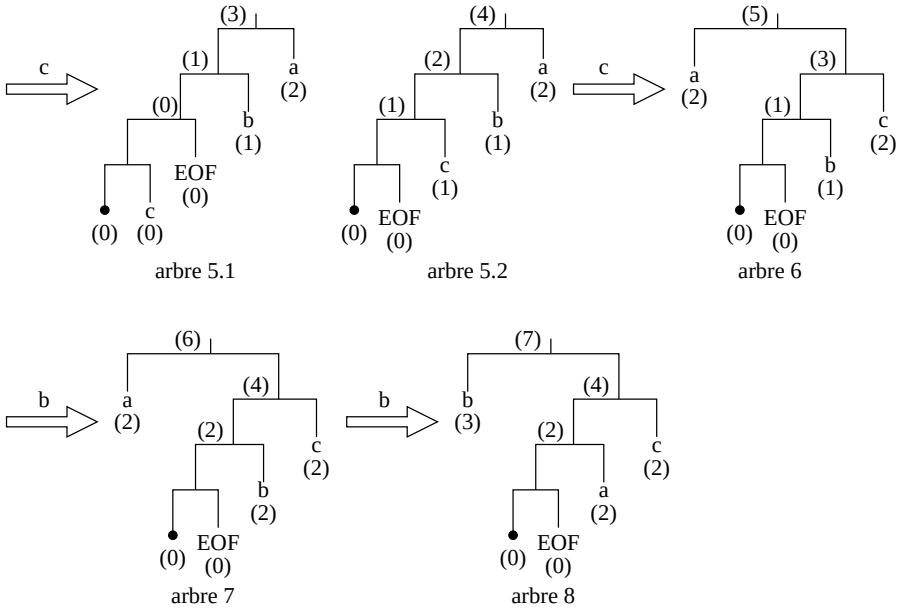
L'arrivée d'un nouveau caractère, «c», nous fait écrire sur le flot de sortie le caractère d'échappement ainsi que son code ASCII : «000|01100011». On insère «c» dans l'arbre (cf. arbre 5.1 ci-dessous) et on effectue la modification de l'arbre (arbre 5.2). On lit à nouveau un caractère «c» sur le flot d'entrée. Comme celui-ci appartient déjà à l'arbre, on n'écrit sur le flot de sortie que son code : «001» et on se contente d'appliquer l'algorithme de modification de l'arbre. On obtient alors l'arbre 6. On lit maintenant un «b» sur le flot d'entrée. On écrit son code sur le flot de sortie «101» et on modifie l'arbre. On obtient alors l'arbre 7. On lit enfin un dernier caractère «b» sur le flot d'entrée. On écrit donc encore une fois le code de «b», qui est encore «101» dans l'arbre 7. On modifie ce dernier et on obtient l'arbre 8. La chaîne est terminée en principe par un EOF. On écrit donc ce dernier : «1001». Finalement, on aura écrit sur le flot de sortie :

$$0|01100001|1|00|01100010|000|01100011|001|101|101|1001,$$

et, si l'on écrit sur disque, comme il y a 44 bits, on rajoute 4 bits pour obtenir 6 octets. On aura donc écrit les octets suivants :

$$00110000\ 11000110\ 00100000\ 11000110\ 01101101\ 10010000.$$

Notons que nous n'avons écrit que 6 octets alors qu'en entrée on avait 8 octets (en comptant EOF).



Pour effectuer la décompression, rien de plus simple : on part comme avec le compresseur de l'arbre 1. Le premier bit lu sur le flot d'entrée est un «0» ce qui signifie que c'est le caractère d'échappement. Par conséquent, on sait que les 8 prochains bits correspondent à un code ASCII. On obtient notre premier «a». On met alors à jour l'arbre de Huffman comme si on avait fait une compression. On a donc maintenant l'arbre 2.2. Le prochain bit lu sur le flot d'entrée est un «1». Donc, en parcourant l'arbre de Huffman 2.2, on voit que cela correspond à un «a». On met alors à jour l'arbre et on obtient l'arbre 3. Et ainsi de suite jusqu'à la fin du fichier. Quand on voit le symbole EOF, on sait que la décompression est finie et on ne lit donc pas les bits qui restent.

2.3.4 Problèmes d'implémentation

Lorsque l'on implémente l'algorithme décrit ci-dessus, il faut choisir un type de données pour stocker les nombres d'occurrences. En principe, on choisit d'utiliser un `unsigned int`. Sur un PC linux, une telle donnée va de 0 à $2^{32} \approx 4,295\ 10^9$. Autrement dit, on peut compresser jusqu'à 4,3Go de fichier sans que le programme plante et commence à faire n'importe quoi. En principe, la plupart des fichiers que l'on compresse ont une taille largement inférieure, donc pas de problème.

Si, d'aventure, on veut compresser des données de plus grande taille, le mécanisme utilisé dans ce cas est en général le suivant : lorsque l'on se rapproche du nombre maximal supporté par les `unsigned int`, on divise tous les nombres d'occurrences de toutes les feuilles par 2 et on recalcule les nombres associés à tous les autres noeuds. C'est ce que l'on appelle en anglais un *rescaling*. On perd un peu de précision dans la manip puisque des nombres d'occurrences aux feuilles/symboles apparus 10 et 11 fois seront tous deux transformés en 5 «apparitions». Cette imprécision se propage évidemment à tous les noeuds de l'arbre. Mais en principe, comme les nombres d'occurrence de tous les noeuds sont relativement grands quand on effectue cette manip, l'imprécision engendrée reste marginale. Notons de plus que cette manipulation ne sera pas fréquente puisqu'elle intervient seulement après avoir lu 4,3Go, 8,6Go, 12,9Go, etc. Ce mécanisme a en fait plus une valeur historique que pratique : en effet, lorsque les architectures étaient de 16 bits, les `unsigned int` s'étalaient de 0 à $2^{16} = 65536$ et il arrivait souvent que les fichiers à compresser aient plus de 65Ko.

Un autre problème qui se pose est la taille du buffer servant à générer les codes écrits sur le flot de sortie. Lorsque l'on reçoit un symbole sur le flot d'entrée, on le recherche dans l'arbre. S'il existe, on récupère son code en parcourant les branches du symbole vers la racine et en inversant l'ordre des bits obtenus. Cela signifie que l'on a besoin d'un buffer pour stocker tout cela. Or, lorsque l'on rajoute de plus en plus de symboles, ce buffer va avoir besoin d'être de plus en plus grand et on risque un *overflow*. Plusieurs solutions existent pour éviter cela : on peut utiliser une liste chaînée pour stocker le buffer, mais c'est en général assez lent. On peut aussi utiliser un `array` et redimensionner ce dernier si besoin est. On peut aussi utiliser un algorithme récursif qui écrit directement les bits dans un fichier (dans ce cas, c'est la pile d'exécution qui va contenir implicitement

l'`array`).

2.4 Quelques variantes de l'algorithme de Huffman

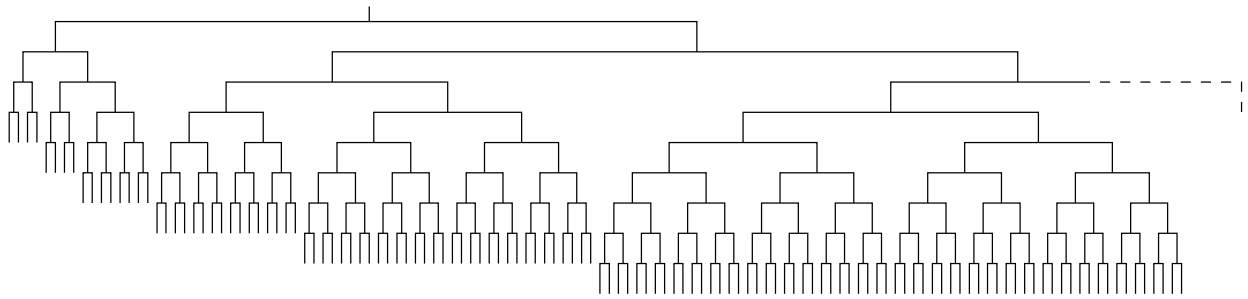
2.4.1 MNP5 et MNP7

Microcom, un fabricant de modem, a développé un protocol appelé MNP (acronyme de Microcom Networking Protocol) afin de spécifier les diverses techniques utilisées pour le transfert de données modem (comment séparer les octets en paquets de bits, comment transmettre des paquets en mode synchrone, asynchrone, etc). Parmi ces spécifications, les classes 5 et 7 de MNP correspondent aux algorithmes de compression. Ces techniques sont utilisées dans la plupart des modems actuels.

Commençons par MNP5. Cette méthode utilise deux étapes :

1. on commence par faire un run length encoding sur les données. Quand au moins 3 octets consécutifs du flot d'entrée sont identiques, on émet sur le flot de sortie 3 fois cet octet, suivi d'un octet servant de compteur de répétition (un 0 indiquant exactement trois répétitions). Par exemple, «AAAAAA» sera codé «AAA3» ;
2. la seconde étape correspond à une variation de l'algorithme de Huffman adaptatif, que nous allons expliciter tout de suite en détails.

La deuxième étape prend en entrée le flot de sortie de la première. Elle pourrait utiliser une méthode de Huffman adaptative, mais les modifications de l'arbre s'avèrent trop coûteuses en temps pour un modem. C'est pourquoi elle utilise un arbre bien particulier (les branches de gauche sont des «0» et les branches de droite des «1») :



Dans tout l'algorithme MNP5, c'est cet arbre qui sera utilisé. Dans Huffman adaptatif, la modification se faisait en intervertissant des noeuds, ce qui bouleversait toute la topologie de l'arbre. Ici, l'idée sera juste d'intervertir des feuilles, ce qui ne bouleversera en rien la topologie du graphe, et ce sera donc beaucoup plus rapide à calculer. L'inconvénient de la méthode est évidemment que la compression réalisée est moins performante que celle de Huffman adaptatif.

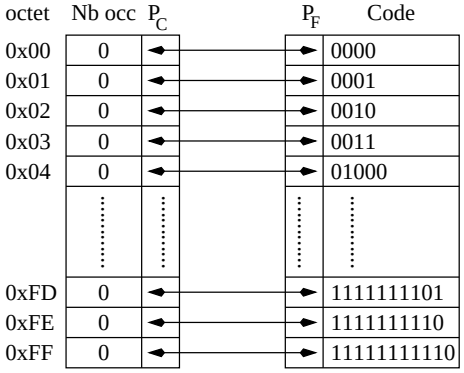
La deuxième étape débute avec le tableau suivant, qui correspond exactement à l'arbre décrit plus haut :

octet	freq	code	octet	freq	code	octet	freq	code	···	octet	freq	code
0x00	0	0000	0x08	0	011000	0x10	0	1000000	···	0xF8	0	1111111000
0x01	0	0001	0x09	0	011001	0x11	0	1000001	···	0xF9	0	1111111001
0x02	0	0010	0x0A	0	011010	0x12	0	1000010	···	0xFA	0	1111111010
0x03	0	0011	0x0B	0	011011	0x13	0	1000011	···	0xFB	0	1111111011
0x04	0	01000	0x0C	0	011100	0x14	0	1000100	···	0xFC	0	1111111100
0x05	0	01001	0x0D	0	011101	0x15	0	1000101	···	0xFD	0	1111111101
0x06	0	01010	0x0E	0	011110	0x16	0	1000110	···	0xFE	0	1111111110
0x07	0	01011	0x0F	0	011111	0x17	0	1000111	···	0xFF	0	11111111110

Lorsqu'un octet a est lu sur le flot d'entrée, le code correspondant dans l'arbre est émis sur le flot de sortie. Comme dans l'algorithme de Huffman adaptatif, le nombre d'occurrences (fréquence) associé à a est incrémenté de 1 et l'on modifie l'arbre pour que les caractères ayant les fréquences les plus élevées aient les codes les plus petits. La différence avec Huffman adaptatif est, qu'ici, on se contentera d'intervertir deux feuilles pour réaliser

cela. Autrement dit, on intervertira juste les codes associés à deux caractères dans la table ci-dessus. Dans MNP5, les nombres d'occurrences sont stockés sur un seul octet. Quand il y a dépassement de capacité, on fait comme dans Huffman adaptatif : on divise tous les nombres d'occurrences par 2.

Afin que la mise à jour du tableau (de l'arbre) soit réalisée rapidement, on génère les deux tableaux ci-dessous :



P_F et P_C sont des pointeurs respectivement vers le tableau contenant les fréquences et vers celui contenant les codes. Ces pointeurs pointent l'un vers l'autre, c'est-à-dire que si $P_F[i]$ pointe sur l'indice j du tableau des fréquences, alors $P_C[j]$ pointe sur l'indice i du tableau des codes. C'est ce qui est symbolisé par les flèches sur la figure ci-dessus. Le tableau des fréquences est classé selon l'ordre croissant des octets, et celui des codes selon l'ordre croissant de leur taille. Par conséquent, si l'on parcourt la table des codes de haut en bas et que l'on créé la liste des fréquences pointées par les codes parcourus, celle-ci est triée par ordre décroissant. Dès lors, l'algorithme de modification de l'arbre est relativement simple : on part du caractère que l'on vient de lire sur le flot d'entrée, supposons qu'il corresponde au nombre i. On incrémente son compteur de fréquence. On regarde alors le code sur lequel il pointe : $P_C[i]$. Si ce code n'est plus valable, c'est que i doit avoir maintenant un code plus court et donc un code qui se trouve au dessus de celui qu'il a actuellement. On regarde le code juste au dessus (indice : $P_C[i]$ - 1). Grâce à P_F, on regarde la fréquence qui lui est associée (indice : $j = P_F[P_C[i] - 1]$). Si celle-ci est supérieure ou égale à celle de Nb_Occ$[i]$, l'arbre n'a pas besoin d'être modifié puisque les codes pointent encore sur des fréquences triées par ordre décroissant. Sinon, on doit intervertir les codes des caractères i et j. Une fois l'intervertion réalisée, on recommence jusqu'à ce que plus aucune intervertion ne puisse arriver. D'où l'algorithme :

```
/* ici, les pointeurs P_F et P_C sont juste les indices des elements */
/* dans les tableaux Nb_Occ et Code */
void Update_Tree(unsigned char i)
{
  unsigned char j;
  unsigned char tmp;

  Nb_Occ[i]++;
  while (P_C[i] > 0 && Nb_Occ[j = P_F[P_C[i]-1]] < Nb_Occ[i])
    {
      tmp = P_C[i]; P_C[i] = P_C[j]; P_C[j] = tmp;
      tmp = P_F[P_C[i]]; P_F[P_C[i]] = P_F[P_C[j]]; P_F[P_C[j]] = tmp;
    }
}
```

MNP7 est un algorithme un peu plus compliqué mais plus efficace. Il utilise lui aussi deux étapes :

1. on commence par faire un run length encoding sur les données. Quand au moins 3 octets consécutifs du flot d'entrée sont identiques, on émet sur le flot de sortie 3 fois cet octet, suivi de 4 bits servant de compteur de répétition (un 0 indiquant exactement trois répétitions) ;
2. la seconde étape correspond à une variation de l'algorithme MNP5.

La différence par rapport à MNP5 est que l'on ne se contente pas d'un arbre dans lequel les feuilles sont les caractères lus, mais on utilise ici un arbre dont les feuilles sont des couples (avant dernier caractère lu, dernier caractère lu). Autrement dit, quand on lit un caractère b sur le flot d'entrée, le code émis sur le flot de sortie dépend du caractère a lu avant lui. Cela correspond, en mathématique, à ce que l'on appelle un modèle de Markov d'ordre 1. L'algorithme consiste donc à créer un arbre similaire à celui de la page précédente, mais pouvant contenir tous les couples de caractères possibles. Supposons qu'on ait lu un caractère a et qu'on lise maintenant un caractère b sur le flot d'entrée. On écrit alors le code correspondant à (a, b) sur le flot de sortie. On incrémente ensuite le nombre d'occurrences de la paire (a, b) et on applique la modification de l'arbre comme dans MNP5. Si on lit maintenant un caractère c, on écrit sur le flot de sortie le code correspondant à la paire (b, c), on incrémente le nombre d'occurrences de cette paire et on effectue la modification de l'arbre, toujours comme dans MNP5. Et ainsi de suite.

L'idée de cet algorithme est que, dans les textes de n'importe quelle langue, la probabilité d'apparition d'un caractère dépend souvent du caractère précédent. Par exemple, si, dans un texte français, l'on trouve un «q», il y a une forte chance que la lettre suivante soit un «u» et très peu de chances que ce soit une autre lettre. Or, dans tous les algorithmes statistiques précédents, on ne tenait pas compte de cette propriété. En effet, si l'on avait rencontré beaucoup de lettres «a» et très peu de «u», on émettait après le code de «q» sur le flot de sortie un long code pour «u». Grâce au modèle de Markov, on émettra après le «q» un code court car le «u» a une très forte chance d'apparaître après le «q».

2.4.2 Le code de Golomb

Les codes dit de *Golomb-Rice* appartiennent à une famille de codes définis pour encoder efficacement des entiers, avec l'hypothèse que plus l'entier est élevé, plus sa probabilité d'apparition est faible. Le code le plus simple, mais pas le plus efficace, est le code unaire. Celui-ci encode l'entier n grâce à n «1», suivis par un «0». Par exemple, 5 est encodé par «111110». Le code unaire correspond à un code de Huffman pour l'alphabet $\{1, 2, 3, \ldots\}$ muni de la loi de probabilité $P(k) = 1/2^k$. Puisque Huffman est optimal pour des puissances négatives de 2 (cf. la sous-section 2.2.2), le code unaire l'est aussi dans ce cas. Cependant, il est très sous-optimal lorsqu'on s'éloigne de cette loi de probabilité. Comme on peut le voir, le code unaire est vraiment très simple à implémenter. En augmentant un petit peu la complexité d'encodage, on trouve un certain nombre de codes séparant l'entier n en deux parties, et représentant la première avec un code unaire et la deuxième avec un autre code. Parmi ceux-ci se trouve le code de Golomb.

Il fut décrit en 1966 dans un petit article (S.W. Golomb, «Run Length Encodings», *IEEE transactions on Information Theory*, IT-12, pp.399–401, juillet 1966), qui commençait ainsi : «Secret agent 00111 is back at the casino again, playing a game of chance, while the fate of mankind hangs in the balance». L'agent 00111 a besoin d'un code pour représenter des séquences où il gagne à la roulette, le code de Golomb est exactement ce dont il a besoin. Il est relativement simple : soit $m > 0$ un entier. Un code de Golomb de paramètre m calcule pour chaque entier n la division euclidienne de n par m, $q = \lfloor n/m \rfloor$, et le reste de la division, $r = n - qm$. Le quotient q est alors encodé grâce à un code unaire. Si m est une puissance de 2, r est encodé sur $\log_2 m$ bits (représentation binaire normale). Si, au contraire, m n'est pas une puissance de 2, on pourrait coder r sur $\lceil \log_2 m \rceil$ bits, mais on peut réduire la longueur du code (et c'est ce que fait le code de Golomb) si l'on utilise une représentation de r sur $\lfloor \log_2 m \rfloor$ bits pour encoder les $2^{\lceil \log_2 m \rceil} - m$ plus petites valeurs de r, et une représentation binaire de $r + 2^{\lceil \log_2 m \rceil} - m$ pour les autres valeurs de r. Le code de Golomb encode l'entier n en écrivant successivement la représentation unaire de q, suivie par la représentation de r mentionnée ci-dessus.

Par exemple, pour $m = 5$,

$$\lceil \log_2 5 \rceil = 3 \qquad \text{et} \qquad \lfloor \log_2 5 \rfloor = 2.$$

Donc, les $8 - 5 = 3$ premières valeurs de r, à savoir 0, 1, 2, seront codées grâce à une représentation de r sur 2 bits, et les 2 dernières valeurs de r, à savoir 3 et 4, seront codées grâce à une représentation binaire sur 3 bits de $r + 3$. Le quotient q, quant à lui, est toujours représenté grâce à un code unaire. Ainsi, pour $n = 12$, $q = 2$ et $r = 2$. q est donc encodé «110», et r «10». Le code de Golomb associe donc au nombre 12 le code «11010». Le tableau ci-après montre les codes des entiers de 0 à 23 :

n	q	r	code	n	q	r	code	n	q	r	code
0	0	0	0|00	8	1	3	10|110	16	3	1	1110|01
1	0	1	0|01	9	1	4	10|111	17	3	2	1110|10
2	0	2	0|10	10	2	0	110|00	18	3	3	1110|110
3	0	3	0|110	11	2	1	110|01	19	3	4	1110|111
4	0	4	0|111	12	2	2	110|10	20	4	0	11110|00
5	1	0	10|00	13	2	3	110|110	21	4	1	11110|01
6	1	1	10|01	14	2	4	110|111	22	4	2	11110|10
7	1	2	10|10	15	3	0	1110|00	23	4	3	11110|110

On peut montrer que le code de Golomb est optimal lorsque la distribution de probabilité des entiers (cf. figure 8 sur la page suivante) est définie par :

$$P(n) = p^{n-1}(1-p), \text{ pour un } p \in]0,1[$$

et lorsque m vérifie :

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil .$$

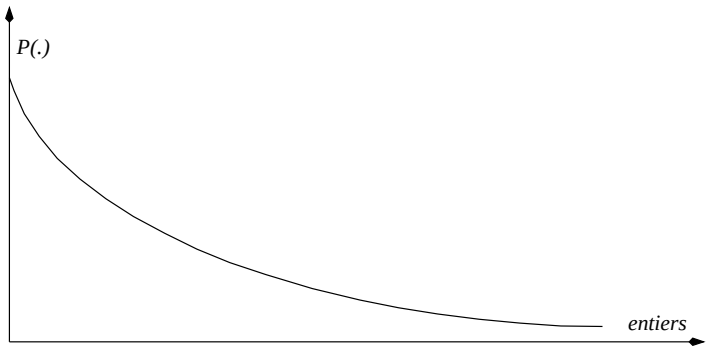


FIG. 8 – La distribution de probabilité des entiers.

Ce code est utilisé entre autres dans l'algorithme de compression conservative JPEG (que nous verrons dans la sous-section 4.1.2).

2.5 Le codage arithmétique

Il est temps de dresser un petit bilan des méthodes statistiques que nous venons de voir : la plus efficace, pour l'instant, est indubitablement la méthode de Huffman. En effet, la méthode de Huffman adaptative est moins performante car ce n'est que lorsqu'elle a lu la majeur partie du flot d'entrée qu'elle affecte les codes optimaux aux différents symboles. MNP5 est encore moins optimale que la méthode de Huffman adaptative puisqu'elle utilise un arbre bien particulier. MNP7 est meilleur, mais c'est, là encore, un algorithme moins performant qu'une méthode de Huffman d'ordre supérieur à 1 (que nous ne verrons pas dans le cadre de ce cours car nous n'avons pas le temps de tout voir).

La méthode de Huffman est donc la plus efficace de toutes celles que nous avons étudiées. Pourtant, elle n'est en général pas optimale. Ici, il convient de rappeler dans quel cadre cette assertion est vraie. Dans le cas où les probabilités d'apparition des symboles sont des puissances de 2, nous avons vu effectivement que Huffman est optimal. Quand les probabilités ne sont pas des puissances de 2, alors Huffman est optimal au sein de la classe d'algorithmes qui, à un symbole donné, associent toujours le même code (par exemple, des algorithmes qui, lorsqu'ils lisent un «a» sur leur flot d'entrée, écriront les bits 1101 sur leur flot de sortie). Par contre, il existe des algorithmes en dehors de cette classe qui donnent de meilleurs résultats que Huffman, c'est-à-dire qui se rapprochent plus de l'entropie. Le codage arithmétique en fait partie.

Huffman est un très bon algorithme, mais il a un gros inconvénient : il code chaque symbole du flot d'entrée sur un nombre entier de bits. Par exemple, si on compresse un message sur un alphabet $\{a_1, a_2, a_3, a_4\}$ et que les probabilités d'apparition des symboles sont :

$$P(a_1) = 0,75 \qquad P(a_2) = 0,1 \qquad P(a_3) = 0,1 \qquad P(a_4) = 0,05,$$

alors Huffman va associer aux symboles de l'alphabet les codes suivants :

$$a_1 \equiv 1 \qquad a_2 \equiv 01 \qquad a_3 \equiv 001 \qquad a_4 \equiv 000.$$

Donc, en moyenne, un symbole est codé sur $0,75 \times 1 + 0,1 \times 2 + 0,1 \times 3 + 0,05 \times 3 = 1,4$ bits. Pourtant, d'après Shannon, un algorithme optimal ne coderait les symboles que sur le nombre de bits suivant :

longueur du code de $a_1 = -\log_2 P(a_1) \approx 0,415$ bits longueur du code de $a_2 = -\log_2 P(a_2) \approx 3,322$ bits
longueur du code de $a_3 = -\log_2 P(a_3) \approx 3,322$ bits longueur du code de $a_4 = -\log_2 P(a_4) \approx 4,432$ bits.

Il en résulterait le codage d'un symbole en moyenne sur $0,75 \times 0,415 + 0,1 \times 3,322 + 0,1 \times 3,332 + 0,05 \times 4,432 \approx 1,197$ bits.

On voit ici ce qui empêche Huffman d'être optimal : Huffman ne peut coder les symboles que sur un nombre entier de bits alors qu'un algorithme optimal devrait les coder sur un nombre fractionnaire de bits. Bien évidemment, il est impossible d'associer à un symbole un code qui ne serait pas constitué d'un nombre entier de bits. C'est pourquoi le codage arithmétique, plutôt que d'associer un code à chaque symbole, associe un code à la totalité du message.

2.5.1 Principe du codage arithmétique

L'idée du codage arithmétique est d'associer un nombre entre 0 et 1 au message. Lorsque l'on débute l'algorithme, on peut avoir n'importe quel message en entrée. L'algorithme représente cette incertitude en spécifiant que le code du message se trouve dans l'intervalle $[0,1[$. Lorsque les symboles sont lus sur le flot d'entrée, ils réduisent l'incertitude sur le contenu du message. L'algorithme représente cette diminution par une réduction de la taille de l'intervalle. Lorsque l'on a lu tous les symboles du flot d'entrée, on choisit comme code du message n'importe quel chiffre entre les bornes du dernier intervalle trouvé. Voyons plus précisément comment l'algorithme fonctionne.

Algorithme 2 *Les différentes étapes du codage arithmétique :*

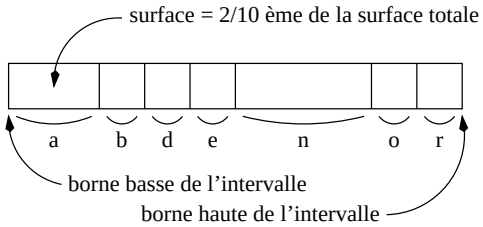
1. *L'intervalle courant est l'intervalle* $[0,1[$. *On détermine les fréquences d'apparition de chacun des symboles de l'alphabet.*
2. *On lit un symbole a sur le flot d'entrée. On effectue une partition de l'intervalle courant en autant de sous-intervalles qu'il y a de symboles dans l'alphabet, et de tailles proportionnelles aux fréquences des symboles.*
3. *On choisit comme nouvel intervalle courant le sous-intervalle correspondant au symbole a.*
4. *S'il reste encore des symboles sur le flot d'entrée, on revient à l'étape 2. Sinon, on écrit sur le flot de sortie n'importe quel nombre compris entre les bornes du nouvel intervalle courant.*

L'algorithme ci-dessus mérite d'être illustré sur un petit exemple :

Exemple 5 : Supposons que nous ayons en entrée le mot «abandonner». À l'étape 1 de l'algorithme, on associe à chaque symbole de l'alphabet sa fréquence :

symbole	fréquence	symbole	fréquence
a	2/10	b	1/10
d	1/10	e	1/10
n	3/10	o	1/10
r	1/10		

Dans toute la suite, on répartira les intervalles de la manière suivante :



Ainsi, l'intervalle $[0,1[$ sera partitionné comme indiqué ci-dessous :

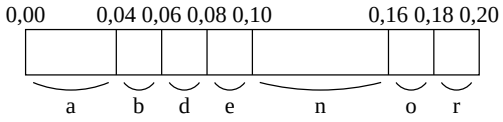
symbole	fréquence	fréquence cumulée	sous-intervalle
a	2/10	02/10	intervalle $[0,0;0,2[$
b	1/10	03/10	intervalle $[0,2;0,3[$
d	1/10	04/10	intervalle $[0,3;0,4[$
e	1/10	05/10	intervalle $[0,4;0,5[$
n	3/10	08/10	intervalle $[0,5;0,8[$
o	1/10	09/10	intervalle $[0,8;0,9[$
r	1/10	10/10	intervalle $[0,9;1,0[$

Passons maintenant à l'étape 2 de l'algorithme. L'intervalle courant est $[0,1[$ et on lit le caractère «a» sur le flot d'entrée. On partitionne alors l'intervalle $[0,1[$ comme indiqué ci-dessus. Étape 3 : on remplace l'intervalle courant par celui correspondant au caractère «a», c'est-à-dire $[0,0;0,2[$. Étape 4 : il reste encore des caractères à lire sur le flot d'entrée, donc on repart à l'étape 2, mais cette fois-ci avec l'intervalle courant $[0,0;0,2[$.

On lit le caractère «b». On partitionne l'intervalle $[0,0;0,2[$ en sous-intervalles de tailles proportionnelles aux fréquences d'apparition des symboles :

symbole	fréquence	sous-intervalle
a	2/10	intervalle $[0,00;0,04[$
b	1/10	intervalle $[0,04;0,06[$
d	1/10	intervalle $[0,06;0,08[$
e	1/10	intervalle $[0,08;0,10[$
n	3/10	intervalle $[0,10;0,16[$
o	1/10	intervalle $[0,16;0,18[$
r	1/10	intervalle $[0,18;0,20[$

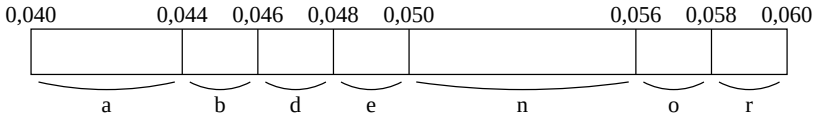
Graphiquement, on obtient bien le découpage annoncé sur la figure ci-dessus :



À l'étape 3, puisque le caractère lu sur le flot d'entrée est un «b», le nouvel intervalle courant est $[0,04;0,06[$. Et c'est avec cet intervalle que l'on repart à l'étape 2. La nouvelle partition de l'intervalle est alors :

symbole	fréquence	sous-intervalle
a	2/10	intervalle $[0,040;0,044[$
b	1/10	intervalle $[0,044;0,046[$
d	1/10	intervalle $[0,046;0,048[$
e	1/10	intervalle $[0,048;0,050[$
n	3/10	intervalle $[0,050;0,056[$
o	1/10	intervalle $[0,056;0,058[$
r	1/10	intervalle $[0,058;0,060[$

On remarque que, là encore, on suit le découpage annoncé au début de l'exemple :



Le nouveau caractère lu sur le flot d'entrée est un «a». Donc le nouvel intervalle courant devient alors $[0,040; 0,044[$. Et ainsi de suite.

Après avoir lu le dernier caractère, le «r», on obtient l'intervalle $[0,0424643292; 0,0424643400[$. On peut alors coder l'ensemble du message en prenant n'importe quel chiffre dans cet intervalle. Bien évidemment, le plus intéressant est de choisir le chiffre qui requiert le moins de bits. ♦

On peut remarquer sur cet exemple la formule qui permet de passer de l'ancien intervalle au nouveau lors de l'étape 3 :

Pour tout symbole a de l'alphabet, soient bas(a) et haut(a) les bornes basses et hautes du sous-intervalle correspondant au symbole a dans l'intervalle $[0,1[$. Soient, de plus, ancienne_borne_basse, ancienne_borne_haute, nouvelle_borne_basse et nouvelle_borne_haute les bornes de l'intervalle courant respectivement au début et à la fin de l'étape 3. Alors :

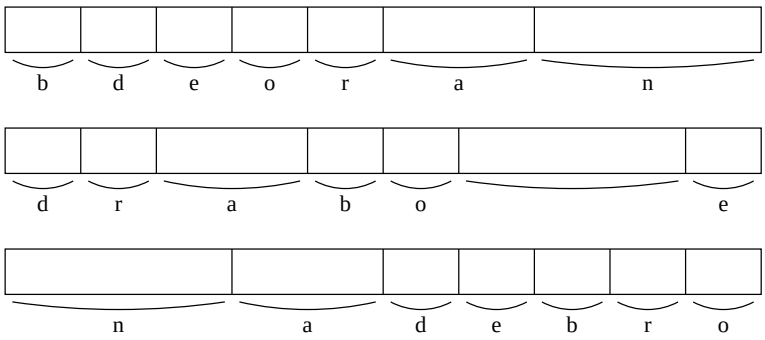
$$\text{nouvelle\_borne\_basse} = \text{ancienne\_borne\_basse} + (\text{ancienne\_borne\_haute} - \text{ancienne\_borne\_basse}) \times \text{bas}(a),$$
$$\text{nouvelle\_borne\_haute} = \text{ancienne\_borne\_basse} + (\text{ancienne\_borne\_haute} - \text{ancienne\_borne\_basse}) \times \text{haut}(a).$$

On peut alors résumer le codage de l'exemple précédent dans le tableau ci-après.

flot d'entrée		bornes des sous-intervalles
a	borne basse	$0,0 + (1,0 - 0,0) \times 0,0 = 0{,}0$
	borne haute	$0,0 + (1,0 - 0,0) \times 0,2 = 0{,}2$
b	borne basse	$0,0 + (0,2 - 0,0) \times 0,2 = 0{,}04$
	borne haute	$0,0 + (0,2 - 0,0) \times 0,3 = 0{,}06$
a	borne basse	$0,04 + (0,06 - 0,04) \times 0,0 = 0{,}04$
	borne haute	$0,04 + (0,06 - 0,04) \times 0,2 = 0{,}044$
n	borne basse	$0,04 + (0,044 - 0,04) \times 0,5 = 0{,}042$
	borne haute	$0,04 + (0,044 - 0,04) \times 0,8 = 0{,}0432$
d	borne basse	$0,042 + (0,0432 - 0,042) \times 0,3 = 0{,}04236$
	borne haute	$0,042 + (0,0432 - 0,042) \times 0,4 = 0{,}04248$
o	borne basse	$0,04236 + (0,04248 - 0,04236) \times 0,8 = 0{,}042456$
	borne haute	$0,04236 + (0,04248 - 0,04236) \times 0,9 = 0{,}042468$
n	borne basse	$0,042456 + (0,042468 - 0,042456) \times 0,5 = 0{,}0424620$
	borne haute	$0,042456 + (0,042468 - 0,042456) \times 0,8 = 0{,}0424656$
n	borne basse	$0,0424620 + (0,0424656 - 0,0424620) \times 0,5 = 0{,}04246380$
	borne haute	$0,0424620 + (0,0424656 - 0,0424620) \times 0,8 = 0{,}04246488$
e	borne basse	$0,04246380 + (0,04246488 - 0,04246380) \times 0,4 = 0{,}042464232$
	borne haute	$0,04246380 + (0,04246488 - 0,04246380) \times 0,5 = 0{,}042464340$
r	borne basse	$0,042464232 + (0,042464340 - 0,042464232) \times 0,9 = 0{,}0424643292$
	borne haute	$0,042464232 + (0,042464340 - 0,042464232) \times 1.0 = 0{,}0424643400$

TAB. 10: codage de l'exemple 5

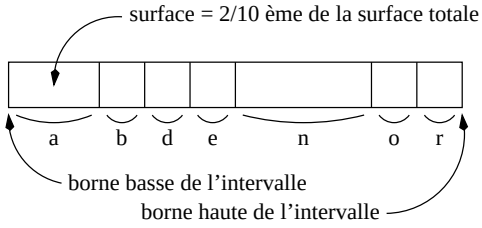
Avant de passer au décodage du message que nous venons de compresser, il convient de préciser que l'ordre dans lequel on place les sous-intervalles dans l'intervalle de départ n'a aucune importance. Par exemple, on aurait très bien pu utiliser pour coder l'exemple 5 les partitions suivantes :



Ce qui est important, c'est que l'encodeur et le décodeur utilisent tous les deux exactement les mêmes partitions. En fait, on pourrait même imaginer que les partitions changent à chaque fois que l'on repasse à l'étape 2. Du moment que l'encodeur et le décodeur effectuent exactement les mêmes changements de partitions, l'algorithme fonctionnera.

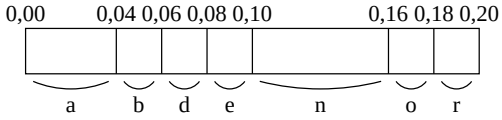
2.5.2 Principe du décodage

Considérons que nous avons à notre disposition le code généré dans la sous-section précédente. Au hasard, j'ai choisi la borne basse du dernier intervalle 0,0424643292. Mais j'insiste sur le fait que n'importe quel nombre entre 0,0424643292 et 0,0424643400 conviendrait. Supposons que, dans l'en-tête du fichier compressé, nous ayons suffisamment d'informations pour reconstituer les partitionnements des intervalles utilisés pour le codage :

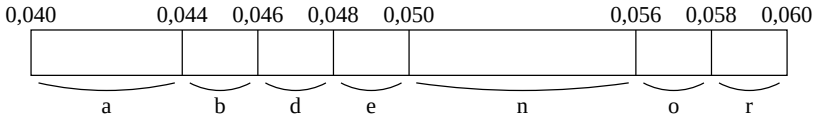


Nous allons maintenant pouvoir procéder au décodage du fichier. Le code de notre message est 0,0424643292. C'est donc un nombre compris entre 0 et 0,2. Par conséquent, comme dans l'intervalle $[0; 1[$, le sous-intervalle $[0; 0,2[$ est affecté à la lettre «a», on sait que le message commence par un «a».

0,0424643292 se trouve à l'intérieur du sous-intervalle $[0,04; 0,06[$ de l'intervalle $[0; 0,2[$, qui correspond à un «b», comme le montre la figure ci-dessous.



En suivant toujours le découpage de la figure ci-dessus, 0,0424643292 appartient au sous-intervalle $[0,040; 0,044[$ de l'intervalle $[0,04; 0,06[$. La troisième lettre du message est donc un «a» d'après la figure ci-dessous.



En fait, on s'aperçoit que le principe de décodage suit exactement ce que faisait l'algorithme de codage. Par symmétrie, on peut donc donner l'algorithme de décodage :

Algorithme 3 *Les différentes étapes du décodage arithmétique :*

1. *L'intervalle courant est l'intervalle* $[0,1[$. *D'après un en-tête, on partitionne l'intervalle* $[0;1[$ *suivant les fréquences d'apparition de chacun des symboles de l'alphabet. On lit sur le flot d'entrée le code C du message.* `Message` *est initialisé à* `vide`.
2. *On regarde à quel sous-intervalle de l'intervalle courant appartient le code. À ce sous-intervalle correspond un symbole, appelons-le a. Alors a est concaténé à* `Message`.
3. *Le sous-intervalle correspondant à a devient le nouvel intervalle courant.*
4. *Si le message n'est pas constitué en entier, on revient à l'étape 2. Sinon,* `Message` *contient l'ensemble du message décodé.*

Comme pour le codage, les étapes 2 et 3 peuvent être effectuées efficacement en remarquant qu'à chaque fois que l'on repasse à l'étape 2, ce que l'on fait, c'est tout simplement regarder à quel sous-intervalle de la partition le code appartient. Supposons que le code soit C et que l'intervalle courant soit $[x;y[$. On sait donc que $C \in [x;y[$, ou encore :

$$\frac{C-x}{y-x} \in [0;1[.$$

Autrement dit, Pour savoir à quel sous-intervalle C appartient, il suffit de lui soustraire la borne inférieure de l'intervalle actuel, de diviser le tout par la taille de l'intervalle et, alors, on peut se reporter au partitionnement pour l'intervalle $[0;1[$. En appliquant récursivement la formule ci-dessus, on peut donc décoder simplement tout le message :

$$\begin{aligned}
(0{,}0424643292 - 0{,}0) \;/\; (1{,}0 - 0{,}0) &= 0{,}0424643292 \in [0;0;0{,}2[&& \Rightarrow \text{a}\\
(0{,}0424643292 - 0{,}0) \;/\; (0{,}2 - 0{,}0) &= 0{,}212321646 \in [0{,}2;0;0{,}3[&& \Rightarrow \text{b}\\
(0{,}212321646 - 0{,}2) \;/\; (0{,}3 - 0{,}2) &= 0{,}12321646 \in [0;0;0{,}2[&& \Rightarrow \text{a}\\
(0{,}12321646 - 0{,}0) \;/\; (0{,}2 - 0{,}0) &= 0{,}6160823 \in [0{,}5;0{,}8[&& \Rightarrow \text{n}\\
(0{,}6160823 - 0{,}5) \;/\; (0{,}8 - 0{,}5) &= 0{,}386941 \in [0{,}3;0{,}4[&& \Rightarrow \text{d}\\
(0{,}386941 - 0{,}3) \;/\; (0{,}4 - 0{,}3) &= 0{,}86941 \in [0{,}8;0{,}9[&& \Rightarrow \text{o}\\
(0{,}86941 - 0{,}8) \;/\; (0{,}9 - 0{,}8) &= 0{,}6941 \in [0{,}5;0{,}8[&& \Rightarrow \text{n}\\
(0{,}6941 - 0{,}5) \;/\; (0{,}8 - 0{,}5) &= 0{,}647 \in [0{,}5;0{,}8[&& \Rightarrow \text{n}\\
(0{,}647 - 0{,}5) \;/\; (0{,}8 - 0{,}5) &= 0{,}49 \in [0{,}4;0{,}5[&& \Rightarrow \text{e}\\
(0{,}49 - 0{,}4) \;/\; (0{,}5 - 0{,}4) &= 0{,}9 \in [0{,}9;1{,}0[&& \Rightarrow \text{r}\\
(0{,}9 - 0{,}9) \;/\; (1{,}0 - 0{,}9) &= 0 && \Rightarrow \text{fin du message}
\end{aligned}$$

Ainsi, les étapes 2 et 3 se ramènent-elles à :

Pour tout symbole a de l'alphabet, soient bas(a) et taille(a) respectivement la borne basse et la taille du sous-intervalle correspondant au symbole a dans l'intervalle $[0,1[$. Appelons C le code courant du message, et [BAS ;HAUT[l'intervalle courant.
Alors le nouveau symbole b à rajouter au message est celui tel que :

$$\frac{C-\text{BAS}}{\text{HAUT}-\text{BAS}} \in [\text{bas}(b);\text{haut}(b)[.$$

On retourne ensuite à l'étape 2 avec, pour nouveau code C, $\frac{C-\text{BAS}}{\text{HAUT}-\text{BAS}}$, et pour nouvel interval courant [bas(b);haut(b)[.

2.5.3 Problème de terminaison de l'algorithme

Jusqu'à maintenant, nous avons passé sous silence le test d'arrêt de l'algorithme de décodage. Problème : comment peut-on savoir que l'on a lu tout le message? Le décodage du message «abandonner» suggère que lorsque le code C devient égal à 0, cela signifie que l'on a tout décodé. En fait, il n'en est rien. D'abord parce que, comme cela avait été indiqué dans la sous-section sur le codage, le code généré à l'issue de la phase de codage peut être n'importe quel nombre entre les bornes du dernier intervalle. Or, on n'a obtenu un 0 à la fin du décodage que parce que le code choisi était précisément la borne basse du dernier intervalle.

La deuxième raison pour laquelle un C à 0 n'indique pas forcément la terminaison de l'algorithme est plus perfide : supposons que nous ayons un alphabet à 3 symboles $\{a_1, a_2, a_3\}$ dont les intervalles respectifs dans $[0; 1[$ sont :

$$a_1 \equiv [0,0; 0,5[\qquad a_2 \equiv [0,5; 0,75[\qquad a_3 \equiv [0,75; 1.0[.$$

Codons maintenant le message $a_3a_2a_1a_1$:

flot d'entrée		bornes des sous-intervalles
a_3	borne basse	$0,0 + (1,0 - 0,0) \times 0,75 = 0,75$
	borne haute	$0,0 + (1,0 - 0,0) \times 1,00 = 1,00$
a_2	borne basse	$0,75 + (1,0 - 0,75) \times 0,50 = 0,875$
	borne haute	$0,75 + (1,0 - 0,75) \times 0,75 = 0,9325$
a_1	borne basse	$0,875 + (0,9325 - 0,875) \times 0,00 = 0,875$
	borne haute	$0,875 + (0,9325 - 0,875) \times 0,50 = 0,90375$
a_1	borne basse	$0,875 + (0,90375 - 0,875) \times 0,00 = 0,875$
	borne haute	$0,875 + (0,90375 - 0,875) \times 0,50 = 0,889375$

On peut maintenant voir le problème qui va se poser si le code représentant le message est la borne inférieure du dernier intervalle de l'algorithme de codage : $0,875$ est la borne de l'intervalle après lecture du deuxième caractère, du troisième et du quatrième caractère. Par conséquent, les codes des messages a_3a_2, $a_3a_2a_1$ et $a_3a_2a_1a_1$ sont identiques. L'algorithme de décodage n'aura donc aucun moyen de savoir s'il a décodé tous les caractères du flot d'entrée ou bien s'il lui en reste encore à écrire. En effet, lorsqu'on décode, on obtient :

$$\begin{aligned}
\text{(0,875 - 0,00) / (1,00 - 0,00)} &= 0,875 \in [0,75; 1,00[&& \Rightarrow a_3 \\
\text{(0,875 - 0,75) / (1,00 - 0,75)} &= 0,5 \in [0,5; 0,75[&& \Rightarrow a_2 \\
\text{(0,5 - 0,5) / (0,75 - 0,50)} &= 0 \in [0,0; 0,5[&& \Rightarrow a_1 \\
\text{(0,0 - 0,0) / (0,50 - 0,00)} &= 0 \in [0,0; 0,5[&& \Rightarrow a_1 \\
\text{(0,0 - 0,0) / (0,50 - 0,00)} &= 0 \in [0,0; 0,5[&& \Rightarrow a_1 \\
\text{(0,0 - 0,0) / (0,50 - 0,00)} &= 0 \in [0,0; 0,5[&& \Rightarrow a_1 \\
\text{(0,0 - 0,0) / (0,50 - 0,00)} &= 0 \in [0,0; 0,5[&& \Rightarrow a_1
\end{aligned}$$

Comment savoir où s'arrêter puisque le code a atteint un point stationnaire qui peut encore signifier la présence du caractère a_1 ?

En fait, rien dans ce que nous avons vu jusqu'à maintenant ne nous permet de déterminer la terminaison de l'algorithme. C'est pourquoi on va rajouter un nouveau symbole, que nous appellerons «EOF» et qui sera le symbole de fin de message. Quand on rencontrera ce caractère, on saura que l'on est arrivé à la fin du message.

2.5.4 Implémentation

Les algorithmes de codage et de décodage sont simples à dérouler sur papier, mais ils posent tout de même un problème du point de vue de l'implémentation : on doit faire des opérations arithmétiques sur des nombres dont la précision est arbitraire. Si l'on compresse de gros fichiers, on peut être amené à avoir besoin de millions de chiffres après la virgule. Évidemment, il est irréaliste de vouloir faire des opérations sur des nombres d'aussi grandes tailles. Un autre inconvénient de la méthode, telle que présentée ci-dessus, est qu'elle fait des opérations sur des nombres réels, or on sait que les opérations sur des flottants sont plus longues à réaliser que sur des entiers et engendrent des pertes de précision. Nous allons voir dans cette sous-section comment pallier ces deux problèmes en même temps. Magique !

Dans la suite, nous allons seulement utiliser deux entiers pour le codage des bornes basses et hautes des intervalles. Pour simplifier l'exposé, nous allons travailler en décimal plutôt qu'en binaire et nous allons supposer que les entiers prennent des valeurs de 0000 à 9999. Bien évidemment, il ne va pas être possible de stocker entièrement les bornes des intervalles sur des entiers aussi courts. Heureusement, on peut constater que, lorsqu'à une étape du codage, les bornes inférieures et supérieures de l'intervalle courant ont quelques uns des chiffres les plus significatifs en commun, alors ces chiffres se retrouveront dans les bornes de tous les intervalles suivants. Par exemple, si l'intervalle courant est $[0,762383405; 0,762483412[$, alors les bornes ont en commun les chiffres les plus significatifs 0,762. Dans ce cas, par la suite, tous les intervalles commenceront par 0,762. Pourquoi ? Simplement parce que, d'après la méthode de construction, les intervalles que l'on construit sont emboités les uns dans les autres. Par conséquent, on est sûr que les intervalles suivants seront au dessus de à 0,7620 et en

dessous de 0,7625. Conclusion : tous les nombres appartenant à ces intervalles commenceront par 0,762, leurs bornes y compris.

Maintenant, rappelons que le code du message correspond à n'importe quel nombre appartenant au dernier intervalle construit. Par conséquent, avant même d'arriver à la fin de la lecture du message, on peut déjà affirmer que son code commencera par 0,762. On peut donc déjà écrire ce nombre sur le flot de sortie. On lit maintenant un nouveau symbole, appelons-le a. D'après les formules données dans la sous-section sur le principe de codage,

$$\begin{aligned}
&\text{nouvelle_borne_basse} = \text{ancienne_borne_basse} + (\text{ancienne_borne_haute} - \text{ancienne_borne_basse}) \times \text{bas}(a),\\
\Leftrightarrow\ &\text{nouvelle_borne_basse} = 0,762383405 + (0,762483412 - 0,762383405) \times \text{bas}(a)\\
\Leftrightarrow\ &\text{nouvelle_borne_basse} = 0,762 + 0,000383405 + (0,000483412 - 0,000383405) \times \text{bas}(a).\\
&\text{nouvelle_borne_haute} = \text{ancienne_borne_basse} + (\text{ancienne_borne_haute} - \text{ancienne_borne_basse}) \times \text{haut}(a),\\
\Leftrightarrow\ &\text{nouvelle_borne_haute} = 0,762 + 0,000383405 + (0,000483412 - 0,000383405) \times \text{haut}(a).
\end{aligned}$$

Ces formules nous montrent que, si l'on possède l'information «les chiffres les plus significatifs sont 0,762», alors les calculs des nouvelles bornes peuvent êtres réalisés uniquement avec les chiffres au delà des millièmes, c'est-à-dire des chiffres moins significatifs que 0,762. Conséquence : on n'a plus besoin des chiffres les plus significatifs pour faire les calculs suivants, on peut donc les éliminer des bornes des intervalles et profiter de cette élimination pour réaliser des décalages des bits de ces bornes vers la gauche de manière à augmenter la précision des calculs.

Exemple : si l'on a l'intervalle $[0,7623; 0,7625[$, comme on code les bornes sur des entiers compris entre 0000 et 9999, ce que l'on code, c'est plutôt : $[7623; 7625[$. Les trois chiffres les plus significatifs sont identiques, donc on va les éliminer et effectuer 3 décalages vers la gauche. 7623 devient alors 3000 (le 3 a été décalé et on a inséré des 0 à sa droite). Pour le 7625, si l'on effectue la même opération, on obtiendra 5000. Ce que l'on fait plutôt, c'est écrire 4999. En effet, il faut penser que les bornes de nos intervalles ont une précision infinie. Or, en mathématiques, $1 = 0,999999999\ldots$ Par conséquent, 4999 auquel on rajoutera autant de 9 que nécessaire lors de nouveaux décalages, est bien égal à 5000. Le problème, si l'on ne fait pas cette petite manipulation, est que l'on risque de ne pas avoir suffisamment de bits dans nos entiers pour pouvoir coder entièrement le message. En effet, supposons qu'on ait obtenu l'intervalle $[0,1111; 0,1112[$, c'est-à-dire les entiers 1111 et 1112. Des décalages en n'insérant que des «0» nous donneraient pour les bornes les nombres 1000 et 2000. Maintenant, supposons que sur l'entrée standard on ne lise plus que le symbole a ayant le sous-intervalle le plus proche de 1 dans la partition de l'intervalle $[0; 1[$. Dans ce cas, à chaque lecture de a, la borne inférieure de l'intervalle augmente, mais la borne supérieure reste inchangée. Les bornes inférieures vont donc graduellement passer de 1000 à 2000, sans jamais atteindre ce nombre. Par conséquent, on ne pourra rien écrire sur le flot de sortie (puisque les chiffres les plus significatifs des bornes inférieures et supérieures seront toujours respectivement 1 et 2).

Exemple 5 (suite) : Reprenons le codage de «abandonner». Au début de l'algorithme, les bornes inférieures et supérieures sont initialisées respectivement à 0000 et 9999, ce qui correspond bien à $[0,0000; 0,9999999\ldots[= [0; 1[$. Lorsqu'on lit le premier «a» sur le flot d'entrée, on doit calculer :

$$\begin{aligned}
\text{nouvelle_borne_basse} &= 0,0 + (1,0 - 0,0) \times \text{bas}(a),\\
\text{nouvelle_borne_haute} &= 0,0 + (1,0 - 0,0) \times \text{haut}(a).
\end{aligned}$$

le $1,0 - 0,0$ correspond à la taille de l'intervalle courant. D'après notre codage, cela doit correspondre à 10000. Lorsque l'on calcule les nouvelles bornes des intervalles, on doit donc utiliser la formule :

$$\begin{aligned}
\text{nouvelle_borne_basse} &= \text{ancienne_borne_basse} + (\text{ancienne_borne_haute} - \text{ancienne_borne_basse} + 0001) \times \text{bas}(a),\\
\text{nouvelle_borne_haute} &= \text{ancienne_borne_basse} + (\text{ancienne_borne_haute} - \text{ancienne_borne_basse} + 0001) \times \text{haut}(a).
\end{aligned}$$

Le 0001 que l'on additionne représente en fait l'infinité de 9 qui devraient se trouver à droite des 4 chiffres significatifs. Les bornes du nouvel intervalle courant après l'arrivée du premier «a» seront donc :

$$\begin{aligned}
\text{nouvelle_borne_basse} &= 0000 + (10000 - 0000) \times 0,0,\\
\text{nouvelle_borne_haute} &= 0000 + (10000 - 0000) \times 0,2.
\end{aligned}$$

Si l'on veut ne réaliser toutes ces opérations que sur des entiers, il convient de représenter 0,0 et 0,2 sur nos entiers à 4 chiffres. $0,2 = 2000/10000$. Donc la formule à appliquer est :

$$\begin{aligned}
\text{nouvelle_borne_basse} &= (0000 \times 10000 + (10000 - 0000) \times 0000)/10000,\\
\text{nouvelle_borne_haute} &= (0000 \times 10000 + (10000 - 0000) \times 2000)/10000.
\end{aligned}$$

On obtient alors pour nouvel intervalle : $[0000; 2000[$, le 2000 étant suivi uniquement par des 0. Il faut préciser ici que la multiplication et la division par 10 ne sont pas douloureuses à effectuer. En effet, cela correspond simplement à faire des décalages respectivement vers la gauche et vers la droite des chiffres. Ce sont donc des opérations à réaliser avec les opérateurs << et >> du langage C. Avant de passer au symbole suivant, il faut encore retirer le dernier bit de la borne supérieure (toujours pour prendre en compte tous les 9 qui suivent le nombre), on a alors l'intervalle : $[0000; 1999[$. Par conséquent, les formules à appliquer sont les suivantes :

$$\begin{aligned}\text{nouvelle\_borne\_basse} &= (0000 \times 10000 + (10000 - 0000) \times 0000)/10000,\\ \text{nouvelle\_borne\_haute} &= (0000 \times 10000 + (10000 - 0000) \times 2000)/10000 - 1.\end{aligned}$$

On lit maintenant le caractère «b» sur le flux d'entrée. On calcule donc les nouvelles bornes :

$$\begin{aligned}&\text{nouvelle\_borne\_basse} = 0,0 + (0,2 - 0,0) \times 0,2,\\ &\text{nouvelle\_borne\_basse} = (0000 \times 10000 + (1999 - 0000 + 0001) \times 2000)/10000 = 0400,\\ &\text{nouvelle\_borne\_haute} = 0,0 + (0,2 - 0,0) \times 0,3\\ &\text{nouvelle\_borne\_haute} = (0000 \times 10000 + (1999 - 0000 + 0001) \times 3000)/10000 - 1 = 0599.\end{aligned}$$

Le nouvel intervalle est alors : $[0400; 0599[$. Hola, stop, vous avez vu ? les bornes inférieure et supérieure ont à gauche un 0 en commun. On va donc envoyer ce 0 sur le flot de sortie et décaler les bornes d'un cran vers la gauche en rajoutant des 0 à droite pour la borne inférieure et des 9 pour la borne supérieure : $[4000; 5999[$.

On lit maintenant un «a» sur le flot d'entrée. On calcule donc les bornes du nouvel intervalle :

$$\begin{aligned}\text{nouvelle\_borne\_basse} &= (4000 \times 10000 + (5999 - 4000 + 0001) \times 0000)/10000 = 4000,\\ \text{nouvelle\_borne\_haute} &= (4000 \times 10000 + (5999 - 4000 + 0001) \times 2000)/10000 - 1 = 4399.\end{aligned}$$

Le nouvel intervalle est donc $[4000; 4399[$. Souvenons-nous que lorsque l'on avait effectué les mêmes opérations dans la sous-section 2.5.1, on avait obtenu l'intervalle $[0,04; 0,044[$. En tenant compte du 0 que l'on a déjà écrit sur le flot de sortie et de l'infinité de 9 à la suite du 4399, on voit bien que les deux intervalles sont identiques.

Les bornes inférieure et supérieure de $[4000; 4399[$ ont à gauche un 4 en commun. Donc on écrit ce 4 sur le flot de sortie et on décale les bornes d'un cran vers la gauche : $[0000; 3999[$. On lit maintenant un «n» sur le flux d'entrée. Calcul du nouvel intervalle :

$$\begin{aligned}\text{nouvelle\_borne\_basse} &= (0000 \times 10000 + (3999 - 0000 + 0001) \times 5000)/10000 = 2000,\\ \text{nouvelle\_borne\_haute} &= (0000 \times 10000 + (3999 - 0000 + 0001) \times 8000)/10000 - 1 = 3199.\end{aligned}$$

On obtient donc l'intervalle $[2000; 3199[$ qui, compte tenu du 04 déjà écrit sur le flot de sortie, correspond bien à l'intervalle $[0,042; 0,0432[$ trouvé dans la sous-section 2.5.1.

Et ainsi de suite. L'ensemble des opérations réalisées par l'algorithme de codage sont consignées dans le tableau 11. On peut ainsi voir que les chiffres 0, 4, 2, 4, 6, 4, sont successivement écrits sur le flot de sortie. À la fin de l'algorithme, il ne nous reste plus qu'à écrire sur le flot de sortie un nombre compris entre les dernières bornes inférieure et supérieure. Si l'on choisit, comme dans la sous-section 2.5.1, d'écrire la borne inférieure, il nous reste à envoyer le nombre 2392. Finalement, on aura écrit sur le flot de sortie la séquence 0424642392. Remarquons que dans la sous-section 2.5.1, nous avions trouvé que le message devait être encodé par 0,0424643292. CQFD. ♦

On peut maintenant formuler précisément l'algorithme de codage :

flot d'entrée		vraies bornes	calcul pratique des bornes		flot de sortie	borne après décalage
a	borne basse	0,0	$(0000 \times 10000 + (9999 - 0000 + 0001) \times 0000)/10000$	$= 0000$		0000
	borne haute	0,2	$(0000 \times 10000 + (9999 - 0000 + 0001) \times 2000)/10000 - 0001$	$= 2000$		1999
b	borne basse	0,04	$(0000 \times 10000 + (1999 - 0000 + 0001) \times 2000)/10000$	$= 0400$	0	4000
	borne haute	0,06	$(0000 \times 10000 + (1999 - 0000 + 0001) \times 3000)/10000 - 0001$	$= 0599$		5999
a	borne basse	0,04	$(4000 \times 10000 + (5999 - 4000 + 0001) \times 0000)/10000$	$= 4000$	4	0000
	borne haute	0,044	$(4000 \times 10000 + (5999 - 4000 + 0001) \times 2000)/10000 - 0001$	$= 4399$		3999
n	borne basse	0,042	$(0000 \times 10000 + (3999 - 0000 + 0001) \times 5000)/10000$	$= 2000$		2000
	borne haute	0,0432	$(0000 \times 10000 + (3999 - 0000 + 0001) \times 8000)/10000 - 0001$	$= 3199$		3199
d	borne basse	0,04236	$(2000 \times 10000 + (3199 - 2000 + 0001) \times 3000)/10000$	$= 2360$	2	3600
	borne haute	0,04248	$(2000 \times 10000 + (3199 - 2000 + 0001) \times 4000)/10000 - 0001$	$= 2479$		4799
o	borne basse	0,042456	$(3600 \times 10000 + (4799 - 3600 + 0001) \times 8000)/10000$	$= 4560$	4	5600
	borne haute	0,042468	$(3600 \times 10000 + (4799 - 3600 + 0001) \times 9000)/10000 - 0001$	$= 4679$		6799
n	borne basse	0,0424620	$(5600 \times 10000 + (6799 - 5600 + 0001) \times 5000)/10000$	$= 6200$	6	2000
	borne haute	0,0424656	$(5600 \times 10000 + (6799 - 5600 + 0001) \times 8000)/10000 - 0001$	$= 6559$		5599
n	borne basse	0,04246380	$(2000 \times 10000 + (5599 - 2000 + 0001) \times 5000)/10000$	$= 3800$		3800
	borne haute	0,04246488	$(2000 \times 10000 + (5599 - 2000 + 0001) \times 8000)/10000 - 0001$	$= 4879$		4879
e	borne basse	0,042464232	$(3800 \times 10000 + (4879 - 3800 + 0001) \times 4000)/10000$	$= 4232$	4	2320
	borne haute	0,042464340	$(3800 \times 10000 + (4879 - 3800 + 0001) \times 5000)/10000 - 0001$	$= 4339$		3399
r	borne basse	0,0424643292	$(2320 \times 10000 + (3399 - 2320 + 0001) \times 9000)/10000$	$= 2392$		2392
	borne haute	0,0424643400	$(2320 \times 10000 + (3399 - 2320 + 0001) \times 10000)/10000 - 0001$	$= 3399$		3399

TAB. 11 – Implémentation du codage arithmétique du message «abandonner».

Algorithme 4 *Au début de l'algorithme, le flot de sortie est vide et l'intervalle courant est* $[0000; 9999[$. *(J'ai choisi ici une représentation sur 4 chiffres décimaux, mais c'est évidemment adaptable à une représentation sur n chiffres binaires).*
À chaque nouveau symbole a sur le flux d'entrée, on effectue les étapes suivantes :

1. *On récupère* $bas(a)$ *et* $haut(a)$ *les bornes du sous-intervalle de* $[0,0;1,0[$ *correspondant au symbole a.* $bas(a)$ *et* $haut(a)$ *sont exprimés dans la représentation à 4 chiffres décimaux. Par exemple, 0,1204 sera représenté par 1204.*
2. *calcul des bornes du nouvel intervalle :*

$$\begin{aligned} nouvelle\_borne\_basse = {} & ancienne\_borne\_basse + \\ & (ancienne\_borne\_haute - ancienne\_borne\_basse + 0001) \times bas(a), \\ nouvelle\_borne\_haute = {} & ancienne\_borne\_basse + \\ & (ancienne\_borne\_haute - ancienne\_borne\_basse + 0001) \times haut(a) - 0001. \end{aligned}$$

3. *Si le chiffre le plus à gauche de la nouvelle borne basse est égal au chiffre le plus à gauche de la nouvelle borne haute, on écrit ce chiffre sur le flot de sortie et on effectue sur les deux bornes un décalage d'un chiffre vers la gauche. On insère alors un 0 dans le chiffre le plus à droite de la borne basse, et un 9 dans celui de la borne haute. On recommence l'étape 3 jusqu'à ce que les chiffres les plus à gauche des deux bornes diffèrent.*
4. *S'il existe encore des symboles à coder, on revient à l'étape 1 avec le nouveau symbole. Sinon, on écrit sur le flot de sortie n'importe quel nombre entre les deux bornes.*

En fait, l'algorithme décrit précédemment marche la plupart du temps, mais il arrive, dans certains cas, qu'il se comporte très mal à cause de ce que les anglais appellent un *buffer underflow*. Voici ce qui risque d'arriver. Soit un alphabet $\{a_1, a_2, a_3, a_4\}$ dont les symboles ont les probabilités et les sous-intervalles décrits dans la table ci-dessous :

symbole	probabilité	sous-intervalle
a_1	0,3600	$[0,6400;1,0000[$
a_2	0,1399	$[0,5001;0,6400[$
a_3	0,1301	$[0,3700;0,5001[$
a_4	0,3700	$[0,0000;0,3700[$

On décide maintenant de coder le message $a_3a_1a_1a_1a_1a_1a_1a_1a_1$. Les étapes de calcul (on effectue les calculs sur ordinateur avec des entiers stockés sur 4 chiffres décimaux) sont décrites dans la table 12. On peut y voir le problème suivant : les bornes inférieures et supérieures se rapprochent de plus en plus au fur et à mesure que l'on rajoute des symboles a_1. Malheureusement, la borne supérieure reste tout le temps à 5000 et la borne inférieure commence par un 4. Par conséquent, on ne peut pas envoyer de chiffres sur le flot de sortie (pour pouvoir le faire il faut que les chiffres les plus à gauche des bornes inf et sup concordent, ce qui n'est pas le cas ici). La borne inférieure se rapproche donc inexorablement de la borne supérieure, mais ce rapprochement s'effectue de plus en plus grâce aux chiffres qui sont au delà des 4 chiffres que l'on utilise pour le stockage de nos entiers. Conséquence : on perd de plus en plus d'informations et, si l'on ne prend pas de contremesures, il va arriver un moment où tous les 4 premiers chiffres des bornes vont se stabiliser à 4999 et 5000, et où seuls les chiffres suivants vont varier. Lorsque cela arrivera, on sera dans une impasse car on ne pourra plus envoyer de chiffres sur le flot de sortie, et on perdra les informations sur tous les nouveaux symboles lus sur le flot d'entrée.

Afin d'éviter ce cas de figure, on va donc rajouter une nouvelle règle. Supposons que les bornes soient respectivement de la forme 49xyz et 50tuv, c'est-à-dire lorsque les deux chiffres les plus à gauche n'ont qu'une unité de différence et que les chiffres suivants sont respectivement des 9 et des 0, alors les deux chiffres les plus significatifs des bornes forment des nombres qui ne diffèrent que d'une unité (ici 49 et 50). De même, si les bornes avaient été 499xy et 500tu, alors 500 - 499 = 1. Par conséquent, il y a très peu d'incertitude sur les prochains chiffres qui seront envoyés sur le flot de sortie. Il n'est donc peut-être pas judicieux d'utiliser dans les entiers stockant les bornes plusieurs chiffres pour conserver aussi peu d'informations. Dans le cas de 499xy et 500tu, lorsque l'on écrira trois chiffres sur le flot de sortie, ce sera soit 499, soit 500. Autrement dit, si l'on commence par écrire un 4, on saura qu'il faudra écrire 2 «9» après, et si l'on commence par écrire un 5, on saura qu'il faudra écrire 2 «0».

symbole		calcul théorique des bornes		bornes sur ordinateur
a_3	borne basse	$0,0+(1,0-0,0)\times 0,3700$	$=0{,}3700$	3700
	borne haute	$0,0+(1,0-0,0)\times 0,5001$	$=0{,}5001$	5000
a_1	borne basse	$0,37+(0,5001-0,37)\times 0,640$	$=0{,}453264$	4532
	borne haute	$0,37+(0,5001-0,37)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,453264+(0,5001-0,453264)\times 0,640$	$=0{,}48323904$	4823
	borne haute	$0,453264+(0,5001-0,453264)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,48323904+(0,5001-0,48323904)\times 0,640$	$=0{,}4940300544$	4940
	borne haute	$0,48323904+(0,5001-0,48323904)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,4940300544+(0,5001-0,4940300544)\times 0,640$	$=0{,}497914819584$	4979
	borne haute	$0,4940300544+(0,5001-0,4940300544)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,497914819584+(0,5001-0,497914819584)\times 0,640$	$=0{,}49931333505024$	4993
	borne haute	$0,497914819584+(0,5001-0,497914819584)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,49931333505024+(0,5001-0,49931333505024)\times 0,640$	$=0{,}4998168006180864$	4998
	borne haute	$0,49931333505024+(0,5001-0,49931333505024)\times 1,000$	$=0{,}5001$	5000
a_1	borne basse	$0,4998168006180864+(0,5001-0,4998168006180864)\times 0,640$	$=0{,}49999048222511104$	4999
	borne haute	$0,4998168006180864+(0,5001-0,4998168006180864)\times 1,000$	$=0{,}5001$	5000

TAB. 12 – Problème d'underflow.

Par conséquent, si l'on crée un compteur `ctr` qui renferme le nombre de chiffres égaux à 0 dans la borne supérieure après le chiffre le plus significatif, et en même temps égaux à 9 dans la borne inférieure, alors on n'a plus besoin de stocker les 0 et les 9. En effet, on sait que si le prochain chiffre à écrire sur le flot de sortie est un 4, on devra écrire `ctr` «9» et sinon `ctr` «0». Exemple : si on a les bornes 499xy et 500tu, alors on va décaler tous les chiffres sauf ceux le plus à gauche de deux cases, ce qui nous donnera 4xy00 et 5tu99, et on affectera à `ctr` la valeur 2. On pourra alors continuer les calculs comme dans le dernier algorithme de codage décrit. Lorsque l'incertitude sur le prochain chiffre à envoyer sur le flot de sortie (un 4 ou un 5) sera levée, on enverra `ctr` chiffres égaux à 0 ou à 9 sur le flot de sortie. Cette petite manipulation nous permet en fait de limiter dans nos entiers le nombre de chiffres utilisés «pour rien». D'où l'algorithme final de codage :

Algorithme 5 *Au début de l'algorithme, le flot de sortie est vide et l'intervalle courant est* $[0000; 9999[$. *On initialise* `ctr` *à 0. À chaque nouveau symbole a sur le flux d'entrée, on effectue les étapes suivantes :*

1. *On récupère $bas(a)$ et $haut(a)$ les bornes du sous-intervalle de $[0,0; 1,0[$ correspondant au symbole a.*
2. *calcul des bornes du nouvel intervalle :*

$$\begin{aligned} nouvelle\_borne\_basse &= ancienne\_borne\_basse + \\ &\quad (ancienne\_borne\_haute - ancienne\_borne\_basse + 0001) \times bas(a), \\ nouvelle\_borne\_haute &= ancienne\_borne\_basse + \\ &\quad (ancienne\_borne\_haute - ancienne\_borne\_basse + 0001) \times haut(a) - 0001. \end{aligned}$$

3. *Si le chiffre le plus à gauche de la nouvelle borne basse est égal au chiffre le plus à gauche de la nouvelle borne haute, on écrit ce chiffre sur le flot de sortie et on effectue sur les deux bornes un décalage d'un chiffre vers la gauche. On insère alors un 0 dans le chiffre le plus à droite de la borne basse, et un 9 dans celui de la borne haute. Si* `ctr` *est différent de 0, alors si la nouvelle borne basse est égale à l'ancienne, on écrit* `ctr` *«9» sinon on écrit* `ctr` *«0», et on affecte 0 à* `ctr`*. On recommence alors à l'étape 3 jusqu'à ce que le chiffre le plus à gauche des deux bornes diffère.*
   Si le chiffre le plus à gauche des bornes ne diffère que d'une unité et que les chiffres suivants sont respectivement des 9 et des 0, alors on incrémente `ctr` *de 1 et on décale tous les chiffres suivants d'une case vers la gauche.*
4. *S'il existe encore des symboles à coder, on revient à l'étape 1 avec le nouveau symbole. Sinon, on écrit sur le flot de sortie n'importe quel nombre entre les deux bornes.*

2.6 Le codage arithmétique adaptatif

Comme pour Huffman, le codage arithmétique dispose d'un algorithme adaptatif. C'est ce dernier que l'on retrouve dans les logiciels du marché (pour des raisons de rapidité de compression). Deux propriétés du codage arithmétique font qu'il peut facilement être rendu adaptatif :

1. Tout d'abord les formules de calcul des bornes des intervalles :

$$\begin{aligned} \text{nouvelle\_borne\_basse} &= \text{ancienne\_borne\_basse} + \\ &\quad (\text{ancienne\_borne\_haute} - \text{ancienne\_borne\_basse} + 0001) \times \text{bas}(a), \\ \text{nouvelle\_borne\_haute} &= \text{ancienne\_borne\_basse} + \\ &\quad (\text{ancienne\_borne\_haute} - \text{ancienne\_borne\_basse} + 0001) \times \text{haut}(a) - 0001. \end{aligned}$$

   Ces formules montrent que, pour encoder un symbole a, il suffit de connaître la fréquence cumulée du symbole a ($\text{bas}(a)$) ainsi que celle du symbole suivant (dans l'intervalle $[0; 1[$). Ceci implique en particulier que la fréquence du symbole a peut être modifiée n'importe quand, du moment que l'encodeur et le décodeur font ces changements de manière synchrone, l'algorithme fonctionnera.
2. L'ordre dans lequel on répartit les symboles dans l'intervalle $[0; 1[$ n'a aucune importance. On peut donc changer cet ordre sans compromettre la validité de l'algorithme (du moment que l'encodeur et le décodeur font ces changements de manière synchrone).

L'idée de l'algorithme adaptatif est la suivante : on lit un nouveau symbole a sur le flot d'entrée. En fonction des fréquences cumulées ($\text{bas}(a)$ et $\text{haut}(a)$) déjà connues, on va déterminer les bornes du nouvel intervalle

courant. On incrémente alors le nombre d'occurrences de a et on remet à jour les différentes fréquences cumulées. On doit effectuer cette mise à jour après le calcul des nouvelles bornes afin de pouvoir synchroniser l'encodeur et le décodeur. On peut alors recommencer avec un nouveau symbole et ainsi de suite.

Bien évidemment, la partie sensible de l'algorithme, celle où l'on risque de passer du temps, c'est la mise à jour des fréquences cumulées. Heureusement, il existe une structure de données qui va nous permettre de l'effectuer très rapidement.

On va stocker les symboles avec leurs nombres d'occurrences et leurs nombres d'occurrences cumulés dans un tableau trié par ordre décroissant de leur nombre d'occurrences. Comme cela, il est plus facile de savoir quelles sont les fréquences cumulées qui ont été modifiées. Afin de pouvoir conserver cet ordre, nous allons utiliser une structure d'arbre binaire balancé (un arbre binaire complet dans lequel il peut manquer certains noeuds en bas à droite). L'arbre aura un noeud pour chacun des symboles et, comme il est bien balancé, sa hauteur sera $\lceil \log_2 n \rceil$, où n est la taille de l'alphabet. Pour $n = 256$, la hauteur de l'arbre sera donc de 8, ce qui devrait limiter les recherches des noeuds dans l'arbre. L'arbre est de plus arrangé de telle sorte que les symboles ayant les plus fortes probabilités vont être placés le plus près de la racine, ce qui, en principe, accélère les temps de recherche des noeuds.

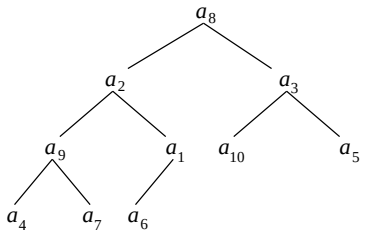
Cette représentation des données semble compliquée au premier abord mais elle est dérisoirement simple à mettre en œuvre en pratique. En effet, considérons un alphabet à 10 symboles dont les nombres d'occurences sont décrits ci-dessous :

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
11	12	12	2	5	1	2	19	12	8

Alors, si l'on retrie ce tableau par ordre décroissant, on obtient :

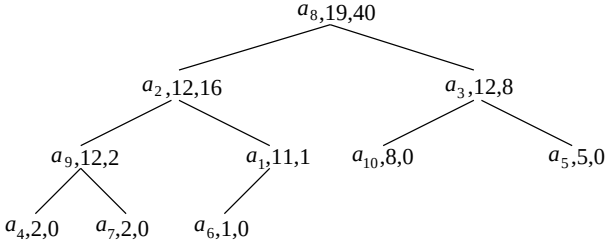
a_8	a_2	a_3	a_9	a_1	a_{10}	a_5	a_4	a_7	a_6
19	12	12	12	11	8	5	2	2	1

Or, on peut voir ce tableau comme un arbre bien balancé. En effet, si l'on considère que le premier élément du tableau correspond à la racine, et que les deux fils du noeud à l'indice i du tableau sont ceux aux indices $2i$ et $2i + 1$, alors on obtient l'arbre bien balancé suivant :



Le noeud a_8 a l'indice 1 dans le tableau. Ses fils, ayant les indices $2 \times 1 = 2$ et $2 \times 1 + 1 = 3$, sont donc les noeuds a_2 et a_3, et ainsi de suite. Lorsque l'on doit remonter vers les parents, rien de plus simple : le parent du noeud d'indice j dans le tableau est le noeud d'indice $\lfloor j/2 \rfloor$.

Afin de simplifier l'algorithme, nous allons stocker en plus du nombre d'occurrences la somme des nombres d'occurrences de tous les symboles du sous-arbre gauche de chacun des noeuds. On obtient ainsi sur la figure ci-dessous les triplets (symbole, nombre d'occurrences, nombre d'occurrences du sous-arbre gauche).



symbole	a_8	a_2	a_3	a_9	a_1	a_{10}	a_5	a_4	a_7	a_6
nombre d'occurrences	19	12	12	12	11	8	5	2	2	1
occurrences du sous-arbre gauche	40	16	8	2	1	0	0	0	0	0

Voyons maintenant comment mettre à jour cette structure lorsqu'un nouveau symbole est lu sur le flot d'entrée. Supposons donc qu'on lise un nouvel a_9. Le nombre d'occurrences de a_9 passe donc de 12 à 13. Pour

retrier le tableau, rien de plus simple : puisque celui-ci est trié par ordre décroissant, il suffit d'intervertir a_9 avec le symbole le plus à gauche dans le tableau (c'est-à-dire de plus petit indice) ayant exactement le même nombre d'occurrences qu'avait a_9, c'est-à-dire 12. On voit ici que l'on doit intervertir a_9 avec a_2. Pour trouver a_2, on peut soit effectuer une recherche linéaire, soit une recherche dichotomique. Après intervertion, on obtient donc le tableau suivant :

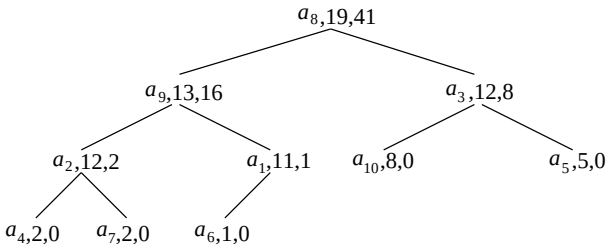
symbole	a_8	a_9	a_3	a_2	a_1	a_{10}	a_5	a_4	a_7	a_6
nombre d'occurrences	19	12	12	12	11	8	5	2	2	1

Comme on s'est contenté de modifier les noms des symboles, il est clair que les nombres d'occurrences des sous-arbres gauches n'ont pas été modifiés par la manip. On obtient donc :

symbole	a_8	a_9	a_3	a_2	a_1	a_{10}	a_5	a_4	a_7	a_6
nombre d'occurrences	19	12	12	12	11	8	5	2	2	1
occurrences du sous-arbre gauche	40	16	8	2	1	0	0	0	0	0

Incrémentons maintenant le nombre d'occurrences de a_9. Dans le tableau, seuls vont augmenter le nombre d'occurrences de a_9 et les nombres des sous-arbres gauches auquel il appartient. Or, faire cette mise à jour est l'enfance de l'art : on part du noeud d'indice j dans le tableau dont le nombre d'occurrences a été augmenté (ici $j = 2$ car a_9 est en deuxième position dans le tableau). On remonte vers son parent (indice $\lfloor j/2 \rfloor$). Si j est pair, alors on incrémente le nombre d'occurrences du sous-arbre gauche de $\lfloor j/2 \rfloor$ de 1, sinon on laisse ce nombre tel qu'il est. On recommence avec pour nouveau j l'indice $\lfloor j/2 \rfloor$, jusqu'à ce qu'on soit arrivé à la racine. On obtient donc le tableau et l'arbre suivants :

symbole	a_8	a_9	a_3	a_2	a_1	a_{10}	a_5	a_4	a_7	a_6
nombre d'occurrences	19	13	12	12	11	8	5	2	2	1
occurrences du sous-arbre gauche	41	16	8	2	1	0	0	0	0	0



Pourquoi modifier le nombre d'occurences du sous-arbre gauche pour l'élément $\lfloor j/2 \rfloor$ du tableau lorsque j est pair et pas lorsque j est impair ? Eh bien tout simplement parce que, d'après la construction de l'arbre, les sous-arbres droits correspondent aux index impairs et les sous-arbres gauches aux index pairs.

La mise à jour de la structure est donc vraiment très simple à effectuer et très rapide (on n'a pas besoin de parcourir inutilement des noeuds du graphe). Voyons maintenant en quoi cette structure nous permet de calculer facilement les bornes des intervalles : nous allons faire un parcours infixe de l'arbre et associer à chaque noeud rencontré un intervalle. Ainsi, on devrait obtenir les intervalles :

symbole	nb occurrence	nb occurrence cumulée	intervalle	intervalle normalisé
a_4	02	02	$[00; 02[$	$[0,000000; 0,023529[$
a_2	12	14	$[02; 14[$	$[0,023529; 0,164706[$
a_7	02	16	$[14; 16[$	$[0,164706; 0,188235[$
a_9	13	29	$[16; 29[$	$[0,188235; 0,341176[$
a_6	01	30	$[29; 30[$	$[0,341176; 0,352941[$
a_1	11	41	$[30; 41[$	$[0,352941; 0,482353[$
a_8	19	60	$[41; 60[$	$[0,482353; 0,705882[$
a_{10}	08	68	$[60; 68[$	$[0,705882; 0,800000[$
a_3	12	80	$[68; 80[$	$[0,800000; 0,941176[$
a_5	05	85	$[80; 85[$	$[0,941176; 1,000000[$

Or il est facile d'obtenir ces intervalles puisque nous avons à notre disposition pour chaque noeud son nombre d'occurrences ainsi que le nombre d'occurrences de son sous-arbre gauche. Pour obtenir l'intervalle correspondant au symbole a dans la quatrième colonne du tableau ci-dessus, il suffit d'appliquer l'algorithme suivant : On part

de la racine et on parcourt l'arbre jusqu'à ce qu'on arrive sur le symbole a. Au début du parcours on initialise une variable `bas` à 0. Chaque fois que l'on part d'un noeud N et que l'on suit une branche de droite, on incrémente `bas` du nombre d'occurrences de N et du nombre d'occurrences du sous-arbre gauche de N. Chaque fois que l'on se déplace sur une branche de gauche, on laisse `bas` inchangé. Lorsque le noeud a est atteint, on ajoute à `bas` le nombre d'occurrences du sous-arbre gauche de a. Il est facile de vérifier que `bas` contient la borne inférieure de a (à une normalisation près). La borne haute s'en déduit immédiatement en rajoutant à `bas` le nombre d'occurrences de a.

Par exemple, pour calculer la borne inférieure de a_{10}, on part de `bas` $= 0$. De la racine on va vers a_3, donc on rajoute à `bas` $19 + 41 = 60$. Arrivé en a_3, on se déplace vers la gauche pour atteindre a_{10}. Déplacement vers la gauche, donc pas de changement de `bas`. On est maintenant arrivé en a_{10} donc on rajoute à `bas` le nombre d'occurrences du sous-arbre gauche de a_{10}, c'est-à-dire 0. La borne inférieure de l'intervalle correspondant à a_{10} est donc 60.

Cet algorithme de calcul des bornes inférieures et supérieures est intéressant car on n'est pas obligé de recalculer toutes les bornes à l'arrivée de chaque symbole. Il suffit de ne recalculer que les bornes du symbole qui vient d'être lu sur le flot d'entrée, d'où un gain de temps appréciable.

2.7 Algorithmes en langage C

2.7.1 Une librairie pour lire/écrire des bits dans des fichiers

```
/* ================================================================================================================ */
/* === CG - prog-bitio.h - version du 1/2/2000                                                                  === */
/* === headers pour pouvoir manipuler des bits dans des fichiers                                                === */
/* ================================================================================================================ */

#ifndef _BITIO_H
#define _BITIO_H


#define READ_BIT_MODE   0  /* flags pour indiquer si l'on a ouvert le fichier en lecture ou en ecriture. */
#define WRITE_BIT_MODE  1  /* Ca permet de tester si les acces en lecture/ecriture sont autorises. */

#define BIT_DATA_LENGTH 65536 /* nombre d'octets dans le buffer de lecture/ecriture */


typedef struct bit_file {
  FILE *file;                          /* pointeur sur le fichier */
  unsigned char access_mode;           /* acces du fichier en lecture ou en ecriture */
  unsigned char data[BIT_DATA_LENGTH]; /* buffer dans lequel sont ecrits les bits */
           int  index;                 /* pour savoir sur quel octet du buffer on est en train de travailler */
  unsigned char mask;                  /* masque permettant d'acceder a un bit de l'octet courant du buffer */
           int  max_index;             /* index du dernier bit d'un fichier ouvert en lecture */
} BIT_FILE;


BIT_FILE *OpenBitFile(char *filename, char *mode);
void      CloseBitFile(BIT_FILE *bit_file);
void      PutBit (BIT_FILE *bit_file, unsigned char bit);
void      PutBits(BIT_FILE *bit_file, unsigned char *code, int count);
int       GetBit (BIT_FILE *bit_file, unsigned char *bit);
int       GetBits(BIT_FILE *bit_file, unsigned char *vect, int count);

#endif /* _BITIO.H */
```

```
/* ================================================================================================================ */
/* === CG - prog-bitio.c - version du 1/2/2000                                                                  === */
/* === routines pour lire/ecrire des sequences de bits sur des fichiers                                         === */
/* ================================================================================================================ */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "prog-bitio.h"
```

```
/* ================================================================================================================ */
/* === Ouverture d'un fichier de bits. En entree, on precise le nom du fichier ainsi que son mode d'acces  === */
/* === ("r" pour une lecture et "w" une ecriture). En retour, on obtient un pointeur sur un BIT_FILE. Si   === */
/* === le fichier n'a pu etre ouvert, le pointeur retourne est a NULL.                                      === */
/* ================================================================================================================ */
BIT_FILE *OpenBitFile(char *filename, char *mode)
{
  /* allocation de la structure permettant de gerer le fichier */
  BIT_FILE *bit_file = (BIT_FILE *) malloc(sizeof(BIT_FILE));
  if (bit_file == NULL) return(NULL);

  /* ouverture du fichier avec le mode d'acces specifie */
  if (strcmp(mode,"r") && strcmp(mode,"w")) return(NULL);
  bit_file->file = fopen(filename, mode);
  if (bit_file->file == NULL) { free((char *)bit_file); return(NULL); }

  /* initialisation de la structure BIT_FILE */
  if (*mode == 'r') bit_file->access_mode = READ_BIT_MODE;
  else bit_file->access_mode = WRITE_BIT_MODE;
  bit_file->index= 0;    /* l'octet courant est le premier du tableau, */
                         /* c'est-a-dire celui qui se trouve le plus a gauche. */
  bit_file->mask = 0x80; /* le premier bit sur lequel on va travailler est celui */
                         /* qui se trouve le plus a gauche dans l'octet courant. */

  /* si le fichier est ouvert en lecture, on lit autant de bits que possible */
  if (bit_file->access_mode == READ_BIT_MODE)
    if ((bit_file->max_index = (unsigned int) fread((char *)bit_file->data,
              1, BIT_DATA_LENGTH, bit_file->file)) != BIT_DATA_LENGTH)
      {
        /* on n'a pas pu lire BIT_DATA_LENGTH octets. Pourquoi? Si ce n'etait pas la fin du fichier, on plante */
        if (!feof(bit_file->file))
          {
            fprintf(stderr,"Erreur fatale (1) dans la fonction OpenBitFile\n");
            fclose(bit_file->file);
            free((char *)bit_file);
            exit(EXIT_FAILURE);
          }
      }

  /* on peut retourner un pointeur sur la structure puisqu'elle a ete bien initialisee */
  return(bit_file);
}


/* ================================================================================================================ */
/* === Fermeture d'un fichier. Avant de fermer le fichier, si ce dernier a ete ouvert en ecriture, on      === */
/* === envoie tous les bits qui trainent dans le buffer sur le fichier.                                    === */
/* ================================================================================================================ */
void CloseBitFile(BIT_FILE *bit_file)
{
  /* on teste quand meme qu'on n'a pas passe une structure vide en parametre */
  if (bit_file == NULL)
    {
      fprintf(stderr,"Erreur fatale (1) dans la fonction CloseBitFile\n");
      exit(EXIT_FAILURE);
    }

  /* si on accedait le fichier en ecriture, on ecrit les derniers bits */
  if (bit_file->access_mode == WRITE_BIT_MODE)
    {
      /* ecriture de tous les octets avant l'octet courant */
      if (bit_file->index > 0)
        if (fwrite((char *)bit_file->data, (int)bit_file->index, 1, bit_file->file) != 1)
          {
            fprintf(stderr,"Erreur fatale (2) dans la fonction CloseBitFile\n");
            fclose(bit_file->file);
            free((char *)bit_file);
            exit(EXIT_FAILURE);
          }

      /* ecriture de l'octet courant si necessaire */
      if (bit_file->mask != 0x80)
        if (fwrite((char *)bit_file->data+bit_file->index,1,1,bit_file->file)!=1)
          {
            fprintf(stderr,"Erreur fatale (3) dans la fonction CloseBitFile\n");
            fclose(bit_file->file);
            free((char *)bit_file);
            exit(EXIT_FAILURE);
          }
```

```
    }

  /* fermeture effective du fichier */
  fclose(bit_file->file);
  free((char *)bit_file);
}


/* ================================================================================================================ */
/* === Ecriture d'un bit dans un fichier. En entree, on passe un pointeur sur un fichier de bits ainsi     === */
/* === qu'un bit (egal a 0 ou 1). Un bit ayant une autre valeur provoquera une erreur fatale et terminera  === */
/* === le programme.                                                                                       === */
/* ================================================================================================================ */
void PutBit(BIT_FILE *bit_file, unsigned char bit)
{
  /* on verifie que la structure du fichier de bits existe bien et que celui-ci a ete ouvert en ecriture */
  if (bit_file == NULL)
    {
      fprintf(stderr,"Erreur fatale (1) dans la fonction PutBit\n");
      exit(EXIT_FAILURE);
    }
  if (bit_file->access_mode != WRITE_BIT_MODE)
    {
      fprintf(stderr,"Erreur fatale (2) dans la fonction PutBit\n");
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* on teste si le bit passe en parametre est bien un 0 ou un 1 */
  if (bit && bit != 1)
     {
      fprintf(stderr,"Erreur fatale (3) dans la fonction PutBit\n");
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* on ecrit le bit dans le buffer */
  if (bit) bit_file->data[bit_file->index] |= bit_file-> mask;
  else bit_file->data[bit_file->index] &= ~bit_file->mask;
  bit_file->mask >>= 1;

  /* on regarde maintenant s'il est temps de changer d'octet dans le buffer */
  /* ou d'ecrire dans le fichier si on est a la fin du buffer */
  if (!bit_file->mask)
    {
      if (bit_file->index == BIT_DATA_LENGTH - 1)
        {
          /* let's go, on ecrit joyeusement dans le fichier tout le buffer */
          if (fwrite((char *)bit_file->data, BIT_DATA_LENGTH, 1, bit_file->file) != 1)
            {
              fprintf(stderr,"Erreur fatale (4) dans la fonction PutBit\n");
              fclose(bit_file->file);
              free((char *)bit_file);
              exit(EXIT_FAILURE);
            }
          /* on remet a jour la structure de donnees */
          bit_file->index= 0;
          bit_file->mask = 0x80;
        }
      else
        {
          bit_file->index++;
          bit_file->mask = 0x80;
        }
    }
}
```

```
/* ================================================================================================================ */
/* === Ecriture de plusieurs bits dans un fichier. En entree, on fournit un pointeur sur le fichier dans   === */
/* === lequel on ecrit, un pointeur sur un vecteur contenant les bits a ecrire, et un entier indiquant le  === */
/* === nombre de bits qu'on souhaite ecrire. On suppose que, dans le vecteur de bits, ceux-ci sont cadres  === */
/* === a gauche, c'est-a-dire que le premier bit a ecrire est le bit le plus a gauche dans le premier      === */
/* === octet du vecteur.                                                                                    === */
/* ================================================================================================================ */
void PutBits(BIT_FILE *bit_file, unsigned char *code, int count)
{
  unsigned char mask;
           int  index;

  /* comme d'habitude, on verifie que le pointeur sur le fichier est valide et qu'il est ouvert en ecriture */
  if (bit_file == NULL)
    {
      fprintf(stderr,"Erreur fatale (1) dans la fonction PutBits\n");
      exit(EXIT_FAILURE);
    }
  if (bit_file->access_mode != WRITE_BIT_MODE)
    {
      fprintf(stderr,"Erreur fatale (2) dans la fonction PutBits\n");
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* on verifie que le nombre de bits a ecrire n'est pas farfelu */
  if  (count <= 0)
    {
      fprintf(stderr,"PutBits : impossible d'ecrire %d bits\n", count);
      fclose(bit_file->file);
      free((char *) bit_file);
      exit(EXIT_FAILURE);
    }

  /* on realise maintenant l'ecriture proprement dite */
  for(mask=0x80, index=0; count; count--)
    {
      if (code[index] & mask) bit_file->data[bit_file->index] |= bit_file-> mask;
      else bit_file->data[bit_file->index] &= ~bit_file->mask;
      bit_file->mask >>= 1;

      /* on regarde maintenant s'il faut changer d'octet dans le buffer */
      /* ou d'ecrire dans le fichier si on est a la fin du buffer */
      if (!bit_file->mask)
        {
          if (bit_file->index == BIT_DATA_LENGTH - 1)
            {
              /* on ecrit dans le fichier tout le buffer */
              if (fwrite((char *)bit_file->data, BIT_DATA_LENGTH, 1, bit_file->file) != 1)
                {
                  fprintf(stderr,"Erreur fatale (3) dans la fonction PutBits\n");
                  fclose(bit_file->file);
                  free((char *)bit_file);
                  exit(EXIT_FAILURE);
                }
              /* on remet a jour la structure de donnees */
              bit_file->index= 0;
              bit_file->mask = 0x80;
            }
          else
            {
              bit_file->index++;
              bit_file->mask = 0x80;
            }
        }
      mask  = (mask == 0x01 ? 0x80 : mask >> 1);
      index = (mask == 0x80 ? index+1 : index);
    }
}
```

```
/* ================================================================================================================ */
/* === Fonction pour lire un bit dans un fichier. Elle prend en parametre un pointeur sur le fichier       === */
/* === contenant les bits ainsi qu'un pointeur sur un caractere. Ce dernier contiendra le bit a la sortie  === */
/* === de la fonction, i.e., il contiendra soit la valeur 0, soit la valeur 1. La fonction renvoie le      === */
/* === nombre de bits lus (0 si on etait deja en fin de fichier. Si le fichier n'existe pas ou n'est pas   === */
/* === accessible en ecriture, on sort violamment du programme.                                            === */
/* ================================================================================================================ */
int GetBit(BIT_FILE *bit_file, unsigned char *bit)
{
  /* on regarde si le fichier passe en parametre est valide et s'il est bien accessible en lecture */
  if (bit_file == NULL)
    {
      fprintf(stderr,"Erreur fatale (1) dans la fonction GetBit\n");
      exit(EXIT_FAILURE);
    }
  if (bit_file->access_mode != READ_BIT_MODE)
    {
      fprintf(stderr,"Erreur fatale (2) dans la fonction GetBit\n");
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* on recupere le bit en question */
  if (bit_file->index == bit_file->max_index)
    /* on essaye d'acceder a un bit qui n'existe pas puisqu'on est deja arrive a la fin du fichier */
    return (0);
  else
    *bit = (bit_file->data[bit_file->index] & bit_file->mask ? 1 : 0);

  /* on pointe sur le bit suivant */
  bit_file->mask >>= 1;
  if (!bit_file->mask)
    {
      bit_file->mask = 0x80;
      bit_file->index++;
    }

  /* on regarde si on doit relire le fichier */
  if (bit_file->index == BIT_DATA_LENGTH)
    {
      bit_file->index = 0;
      if ((bit_file->max_index = (unsigned int) fread((char *)bit_file->data,
                1, BIT_DATA_LENGTH, bit_file->file)) != BIT_DATA_LENGTH)
        {
          /* on n'a pas pu lire BIT_DATA_LENGTH octets. Pourquoi ? */
          /* si ce n'etait pas la fin de fichier, on plante */
          if (!feof(bit_file->file))
            {
              fprintf(stderr,"Erreur fatale (4) dans la fonction GetBit\n");
              fclose(bit_file->file);
              free((char *)bit_file);
              exit(EXIT_FAILURE);
            }
        }
    }

  /* on indique qu'on a bien lu un bit */
  return(1);
}


/* ================================================================================================================ */
/* === fonction pour lire plusieurs bits dans un fichier. On passe en parametre un pointeur sur le         === */
/* === fichier de bits, un pointeur sur le vecteur (deja alloue en memoire) qui va contenir les bits, et   === */
/* === un entier qui represente le nombre de bits a lire. La fonction retourne le nombre de bits qu'elle   === */
/* === a reussi a lire.                                                                                    === */
/* ================================================================================================================ */
int GetBits(BIT_FILE *bit_file, unsigned char *vect, int count)
{
  unsigned char mask;
           int  index;
           int  nb_bits_read;

  /* on verifie que le pointeur sur le fichier est valide et que le fichier est bien accede en lecture */
  if (bit_file == NULL)
    {
      fprintf(stderr,"Erreur fatale (1) dans la fonction GetBits\n");
      exit(EXIT_FAILURE);
```

```
    }
  if (bit_file->access_mode != READ_BIT_MODE)
    {
      fprintf(stderr,"Erreur fatale (2) dans la fonction GetBits\n");
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* on verifie que le nombre de bits qu'on veut lire n'est pas farfelu */
  if  (count <= 0)
    {
      fprintf(stderr,"GetBits : impossible de lire %d bits\n", count);
      fclose(bit_file->file);
      free((char *)bit_file);
      exit(EXIT_FAILURE);
    }

  /* lecture proprement dit */
  for(mask = 0x80, index = 0, nb_bits_read = 0; count;
      count--, mask = (mask == 0x01 ? 0x80 : mask >> 1), index = (mask == 0x80 ? index+1 : index))
    {
      /* on regarde si on peut recuperer le bit courant, ou bien si on est deja arrive en fin de fichier */
      if (index == bit_file->max_index)
        /* on est bien arrive en fin de fichier */
        return nb_bits_read;

      /* on recupere le bit courant */
      if (bit_file->data[bit_file->index] & bit_file->mask) vect[index] |= mask;
      else vect[index] &= ~mask;
      nb_bits_read++;

      /* on pointe sur le bit suivant */
      bit_file->mask >>= 1;
      if (!bit_file->mask)
        {
          bit_file->mask = 0x80;
          bit_file->index++;
        }

      /* on regarde si on doit relire le fichier */
      if (bit_file->index == BIT_DATA_LENGTH)
        {
          bit_file->index = 0;
          if ((bit_file->max_index = (unsigned int) fread((char *)bit_file->data,
                   1, BIT_DATA_LENGTH, bit_file->file)) != BIT_DATA_LENGTH)
            {
              /* on n'a pas pu lire BIT_DATA_LENGTH octets. Pourquoi ? */
              /* si ce n'etait pas la fin de fichier, on plante */
              if (!feof(bit_file->file))
                {
                  fprintf(stderr,"Erreur fatale (3) dans la fonction GetBis\n");
                  fclose(bit_file->file);
                  free((char *)bit_file);
                  exit(EXIT_FAILURE);
                }
            }
        }
    }
  /* on retourne le nombre de bits lus */
  return (nb_bits_read);
}
```

2.7.2 Algorithme de Huffman

```
/* ================================================================================================================ */
/* === CG - prog-sec2-huffman.c - version du 5/8/2001                                                           === */
/* === realise l'algorithme de Huffman tel que decrit dans la section 2.                                        === */
/* ================================================================================================================ */

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "prog-bitio.h"


#define READ_LENGTH 65536 /* le nombre d'octets lus a la fois dans le fichier */
                          /* a compresser (valable pour les 2 passes d'Huffman) */
```

```
/* ================================================================================================================ */
/* === definition d'une structure pour les arbres binaires a la Huffman                                         === */
/* ================================================================================================================ */
typedef struct _Arbre Arbre;

struct _Arbre {
  Arbre *parent;
  Arbre *fils_gauche;
  Arbre *fils_droit;
  unsigned int freq;
};


/* ================================================================================================================ */
/* === definition d'une structure pour stocker les codes des caracteres.                                        === */
/* ================================================================================================================ */
typedef struct {
  unsigned char *code;  /* vecteur de bits du code */
  int            count; /* nombre de bits dans le code */
} Code;


/* ================================================================================================================ */
/* === la fonction ci-dessous prend en entree le nom du fichier a compresser. Elle lit le fichier et           === */
/* === stocke les frequences d'apparition de chacun des caracteres dans un tableau 'freq' qu'elle alloue       === */
/* === et elle renvoie ce tableau. En plus des 256 cases utilisees pour stocker les frequences de chaque       === */
/* === caractere, une 257eme est rajouteea la fin du tableau, qui contient le nombre de caracteres que         === */
/* === contient l'ensemble du fichier (modulo le nombre d'unsigned int possibles).                              === */
/* ================================================================================================================ */
unsigned int *cree_frequences(char *filename)
{
  unsigned int *freq;
  FILE         *in;
  unsigned char buffer[READ_LENGTH];
  int           i, j;
  size_t        nb_read;

  /* creation du tableau freq */
  freq = (unsigned int *) malloc(257 * sizeof(unsigned int));
  if (freq == NULL)
    {
      fprintf(stderr, "cree_frequences: allocation memoire impossible\n");
      exit(EXIT_FAILURE);
    }

  /* ouverture du fichier a compresser */
  in = fopen(filename, "r");
  if (in == NULL)
    {
      free((char *)freq);
      fprintf(stderr, "cree_frequences: impossible d'ouvrir le fichier %s\n", filename);
      exit(EXIT_FAILURE);
    }

  /* initialisation du tableau */
  for (i=0; i<256; i++) freq[i] = 0;

  /* mise a jour du tableau freq */
  while ((nb_read = fread((void *)buffer, 1, READ_LENGTH, in)) != 0)
    for (i=0; i<nb_read; i++)
      {
        if (freq[(int)buffer[i]] == UINT_MAX)
          /* on a atteint la frequence max => on divise toutes les frequences par 2 */
          for (j=0; j<256; j++) freq[j] /= 2;

        freq[(int)buffer[i]]++;
        freq[256]++;
      }

  /* fermeture du fichier */
  fclose(in);

  return freq;
}
```

```
/* ================================================================================================================ */
/* === la fonction ci-dessous prend en argument deux elements de type Arbre et renvoie -1 si le premier    === */
/* === element a une frequence strictement inferieure au deuxieme element, 0 si elles sont egales et 1     === */
/* === dans le cas restant.                                                                                 === */
/* ================================================================================================================ */
int compare_freq(const void *elt1, const void *elt2)
{
  unsigned int nb1, nb2;

  if ((elt1 == NULL) || (elt2 == NULL))
    {
      fprintf(stderr, "compare_freq: impossible de comparer un element NULL\n");
      exit(EXIT_FAILURE);
    }

  nb1 = (*(Arbre **)elt1)->freq; nb2 = (*(Arbre **)elt2)->freq;
  if (nb1 < nb2) return -1;
  if (nb1 > nb2) return  1;
  return 0;
}


/* ================================================================================================================ */
/* === la fonction ci-dessous prend en argument un tableau de frequences et cree un arbre de huffman       === */
/* === correspondant. Un tableau contenant pour chacun des 256 caracteres possibles, soit sa feuille dans  === */
/* === l'arbre, soit un noeud vide (pas de parent, pas d'enfant et une frequence a 0) si le caractere      === */
/* === n'apparait pas dans le texte, est alors renvoye par la fonction.                                     === */
/* ================================================================================================================ */
Arbre *cree_arbre(unsigned int freq[256])
{
  Arbre *tmp_tab[256], *new_arb, **tab, *arb;
  int    i, trouve, nb_car = 256;
  int    nb_avant_new_arb, nb_apres_new_arb;

  /* creation des feuilles de l'arbre : a chaque caractere est associe un sous-arbre (pour l'instant vide) */
  arb = (Arbre *) malloc(256 * sizeof(Arbre));
  if (arb == NULL)
    {
      fprintf(stderr, "cree_arbre: allocation memoire impossible\n");
      exit(EXIT_FAILURE);
    }
  for (i=0; i<nb_car; i++)
    {
      arb[i].parent = NULL;
      arb[i].fils_gauche = NULL; arb[i].fils_droit = NULL;
      arb[i].freq = freq[i];
    }

  /* on cree un tableau associant a chaque caractere sa frequence. et on trie ce */
  /* tableau par ordre croissant des frequences */
  for (i=0; i<nb_car; i++) tmp_tab[i] = arb + i;
  qsort(tmp_tab, nb_car, sizeof(Arbre *), compare_freq);

  /* on ne cree l'arbre que pour les frequences strictement positives */
  for (tab = tmp_tab; nb_car > 0 && tab[0]->freq == 0; nb_car--, tab++);
  if (tab[0]->freq == 0)
    {
      printf("cree_arbre: impossible de compresser un fichier vide\n");
      exit(EXIT_FAILURE);
    }

  /* creation de l'arbre de Huffman a partir des frequences */
  for (; nb_car >= 2; nb_car--)
    {
      /* creation d'un noeud regroupant les deux caracteres de plus faibles frequences */
      new_arb = (Arbre *) malloc(sizeof(Arbre));
      if (new_arb == NULL)
        {
          fprintf(stderr, "cree_arbre: allocation memoire impossible\n");
          exit(EXIT_FAILURE);
        }
      tab[0]->parent = new_arb;      tab[1]->parent = new_arb;
      new_arb->fils_gauche = tab[0]; new_arb->fils_droit  = tab[1];
      new_arb->parent = NULL;

      /* affectation de la frequence a new_arb. Si celle-ci depasse la taille des */
      /* unsigned int, on divise toutes les frequences par 2 */
      if (tab[0]->freq + tab[1]->freq < tab[0]->freq)
        for (i=0; i<nb_car; i++) tab[i]->freq /= 2;
```

```
      new_arb->freq = tab[0]->freq + tab[1]->freq;

      /* on va maintenant eliminer les deux premiers elements de tab et inserer */
      /* new_tab de maniere a ce que tab soit trie par ordre croissant de frequence */
      for (i=2, trouve=0; i<nb_car && !trouve; i++)
        if (tab[i]->freq >= new_arb->freq) trouve++;
      nb_avant_new_arb = i-2; /* nb d'elts de tab qui vont etre deplaces de 2 cases pour que tab reste trie */
      if (nb_avant_new_arb) memmove(tab, tab+2, nb_avant_new_arb * sizeof(Arbre *));
      tab[nb_avant_new_arb] = new_arb;
      nb_apres_new_arb = nb_car - i;
      if (nb_apres_new_arb) memmove(tab+i-1, tab+i, nb_apres_new_arb * sizeof(Arbre *));
    }

  /* on retourne l'arbre correspondant a l'ensemble des caracteres */
  return arb;
}
/* ================================================================================================================ */
/* === la fonction suivante prend un arbre de Huffman et associe a chaque caractere son code d'apres       === */
/* === l'arbre passe en parametre (les fils gauches ont un bit a zero et les fils droits un bit a 1).      === */
/* ================================================================================================================ */
Code *cree_code(Arbre arb[256])
{
  unsigned char mask;
  Code         *code;
  int           i, j, k;
  Arbre        *tmp_arb;


  /* creation du tableau de codes */
  code = (Code *) malloc(256 * sizeof(Code));
  if (code == NULL)
    {
      fprintf(stderr, "cree_code: allocation memoire impossible\n");
      exit(EXIT_FAILURE);
    }

  /* creation des vecteurs de bits pour chaque caractere */
  for (i=0; i<256; i++)
    if (arb[i].freq)
      {
        /* on calcule la longueur du vecteur de bit */
        for (j=0, tmp_arb=arb+i; tmp_arb->parent!=NULL; j++, tmp_arb=tmp_arb->parent);

        /* creation du vecteur de bits */
        if (j == 0)
          {
            /* j = 0 signifie que le fichier ne contient qu'un seul caractere */
            /* => on peut lui affecter juste 1 bit a 0 */
            code[i].code = (unsigned char *)malloc(sizeof(unsigned char));
            if (code[i].code == NULL)
              {
                fprintf(stderr, "cree_code: allocation memoire impossible\n");
                exit(EXIT_FAILURE);
              }
            code[i].code[0] = 0;
            code[i].count   = 1;
          }
        else
          {
            /* ici, on doit creer un vecteur de longueur le nombre d'arcs */
            /* separant la feuille/caractere de la racine de l'arbre */
            code[i].code = (unsigned char *)malloc(((j+7)/8) * sizeof(unsigned char));
            if (code[i].code == NULL)
              {
                fprintf(stderr, "cree_code: allocation memoire impossible\n");
                exit(EXIT_FAILURE);
              }

            /* remplissage des bits */
            for (k=0; k<(j+7)/8; k++) code[i].code[k] = (unsigned char) 0;
            code[i].count = j;
            for (tmp_arb = arb+i, j--, k=j/8, mask=0x80 >> (j%8); j>=0;
                 tmp_arb = tmp_arb->parent, j--, mask=(mask==0x80 ? 0x01 : mask << 1), k=(mask==0x01 ? k-1 : k))
              if (tmp_arb->parent->fils_droit == tmp_arb) code[i].code[k] |= mask;
          }
      }
    else
      {
        code[i].code = NULL;
```

```
        code[i].count = 0;
      }

  /* on renvoie les codes */
  return code;
}


/* ================================================================================================================ */
/* === la fonction suivante ouvre un fichier en lecture. Elle calcule un arbre de Huffman correspondant    === */
/* === ainsi qu'un codage coherent avec cet arbre. Enfin, elle utilise ce codage pour compresser le        === */
/* === fichier ouvert en lecture. Le fichier de sortie a un en-tete de 7 caracteres egaux a la chaine      === */
/* === "Huffman", suivi par un tableau de 256 unsigned int contenant les frequences des caracteres, suivi  === */
/* === par le nombre d'octets dans le fichier d'origine (modulo le nombre total de unsigned int            === */
/* === possibles), enfin suivi par les octets compresses.                                                   === */
/* ================================================================================================================ */
void compresse(char *filein, char *fileout)
{
  FILE         *in;
  BIT_FILE     *out;
  unsigned char buffer[READ_LENGTH];
  int           i, nb_read;
  Arbre        *arb;
  unsigned int *freq;
  Code         *code;

  /* calcul des frequences, de l'arbre de Huffman et des codes */
  freq = cree_frequences(filein);
  arb  = cree_arbre(freq);
  code = cree_code(arb);

  /* ouverture des fichiers d'entree et de sortie en vue de l'ecriture */
  in = fopen(filein, "r");
  if (in == NULL)
    {
      fprintf(stderr, "compresse: impossible d'ouvrir le fichier %s\n", filein);
      exit(EXIT_FAILURE);
    }
  out = OpenBitFile(fileout, "w");
  if (out == NULL)
    {
      fclose(in);
      fprintf(stderr, "compresse: impossible d'ouvrir le fichier %s\n", fileout);
      exit(EXIT_FAILURE);
    }

  /* ecriture de l'ent-tete */
  PutBits(out, "Huffman", 56);
  PutBits(out, (unsigned char *)freq, 257 * sizeof(unsigned int) * 8);

  /* ecriture des caracteres dans le fichier compresse */
  while ((nb_read = fread((void *)buffer, 1, READ_LENGTH, in)) != 0)
    for (i=0; i<nb_read; i++) PutBits(out, code[(int) buffer[i]].code, code[(int) buffer[i]].count);

  /* fermeture des fichiers */
  fclose(in);
  CloseBitFile(out);
}


/* ================================================================================================================ */
/* === la fonction suivante ouvre un fichier en lecture, recupere les frequences du fichier d'origine,     === */
/* === calcule un arbre de Huffman correspondant, puis utilise celui-ci pour decompresser le fichier       === */
/* === ouvert en lecture. La decompression est ecrite dans le fichier "fileout". La fonction teste bien    === */
/* === entendu si l'en-tete du fichier d'entree est correct (cf. la fonction compresse) avant de realiser  === */
/* === les operations de decompression.                                                                     === */
/* ================================================================================================================ */
void decompresse(char *filein, char *fileout)
{
  BIT_FILE      *in;
  FILE          *out;
  unsigned char  bit;
  unsigned int   freq[256], nb_octets_fichier, total_octets_lus = 0;
  char           en_tete[8];
  Arbre         *arb, *root, *tmp_arb;
  int            i, trouve;

  /* ouverture des deux fichiers */
  in = OpenBitFile(filein, "r");
```

```
if (in == NULL)
  {
    fprintf(stderr, "decompresse: impossible d'ouvrir le fichier %s\n", filein);
    exit(EXIT_FAILURE);
  }
out = fopen(fileout, "w");
if (out == NULL)
  {
    CloseBitFile(in);
    fprintf(stderr, "decompresse: impossible d'ouvrir le fichier %s\n", fileout);
    exit(EXIT_FAILURE);
  }

/* on recupere l'en-tete du fichier d'entree */
GetBits(in, en_tete, 56); en_tete[7] = '\0';
if (strcmp(en_tete, "Huffman"))
  {
    fprintf(stderr, "decompresse: ce fichier n'a pas ete compresse avec Huffman\n");
    fclose(out);
    CloseBitFile(in);
    exit(EXIT_FAILURE);
  }

/* recuperation des frequences, de l'arbre de Huffman et du codage */
if (GetBits(in,(char *)freq, 256*sizeof(unsigned int)*8) != 256*sizeof(unsigned int)*8)
  {
    fclose(out);
    CloseBitFile(in);
    fprintf(stderr, "decompresse: ce fichier n'a pas ete compresse avec Huffman (2)\n");
    exit(EXIT_FAILURE);
  }
arb  = cree_arbre(freq);

/* recuperation du nombre d'octets dans le fichier d'origine */
if (GetBits(in, (char *) &nb_octets_fichier, sizeof(unsigned int) * 8) != sizeof(unsigned int) * 8)
  {
    fclose(out);
    CloseBitFile(in);
    fprintf(stderr, "decompresse: ce fichier n'a pas ete compresse avec Huffman (3)\n");
    exit(EXIT_FAILURE);
  }

/* on recupere la racine de l'arbre de Huffman */
for (i=0, trouve=0; i<256 && !trouve; i++) if (arb[i].freq) trouve ++;
for (root=arb+i-1; root->parent != NULL; root = root->parent);

/* maintenant on decompresse vraiment le fichier d'entree */
if (root == arb+i-1)
  /* il n'y a qu'un seul caractere dans tout le fichier */
  while(GetBit(in, &bit) && total_octets_lus != nb_octets_fichier)
    {
      total_octets_lus++;
      fprintf(out, "%c", (char) (root - arb));
    }
else
  /* ici, on a un fichier "normal", i.e., contenant plusieurs caracteres differents */
  while(GetBit(in, &bit) && total_octets_lus != nb_octets_fichier)
    {
      total_octets_lus++;
      if (bit) tmp_arb = root->fils_droit;
      else tmp_arb = root->fils_gauche;

      for (; tmp_arb->fils_gauche && tmp_arb->fils_droit;)
        if (GetBit(in, &bit))
          {
            if (bit) tmp_arb = tmp_arb->fils_droit;
            else tmp_arb = tmp_arb->fils_gauche;
          }
        else
          {
            fclose(out);
            CloseBitFile(in);
            fprintf(stderr, "ce fichier n'a pas ete compresse avec Huffman (4)\n");
            exit(EXIT_FAILURE);
          }
     fprintf(out, "%c", (char) (tmp_arb - arb));
    }

/* fermeture des deux fichiers */
```

```
  fclose(out);
  CloseBitFile(in);
}


/* ================================================================================================================ */
/* === le programme principal attend trois arguments. Le premier indique si l'on doit compresser ou        === */
/* === decompresser un fichier. le second est le nom du fichier sur lequel va porter l'action, et le       === */
/* === troisieme est le nom du fichier dans lequel sera stocke le resultat de l'action.                    === */
/* ================================================================================================================ */
int main(int argc, char **argv)
{
  /* test des arguments */
  if ((argc != 4) || (strcmp(argv[1], "compresse") && strcmp(argv[1], "decompresse")))
    {
      fprintf(stderr, "usage : prog-sec2-huffman action fic_entree fic_sortie\n");
      fprintf(stderr, "        action doit etre egal soit a la chaine compresse soit a decompresse\n");
      exit (EXIT_FAILURE);
    }

  /* on effectue l'action demandee */
  if (!strcmp(argv[1], "compresse")) compresse(argv[2], argv[3]);
  else decompresse(argv[2], argv[3]);

  return EXIT_SUCCESS;
}
```

2.8 Exercices

Exercice 1 On recherche un nombre entre 1 et 2^N. Un certain nombre d'algorithmes sont conçus pour cela. Parmi ceux-ci figure le célèbre algorithme par dichotomie. Montrez que ce dernier est, en moyenne, optimal, c'est-à-dire que, d'une manière générale, c'est l'algorithme qui permet de retrouver le nombre recherché en un minimum de temps.

Exercice 2 On sait qu'un texte sur un alphabet $\{a_1, \ldots, a_4\}$ vérifie les probabilités suivantes :

$$
\begin{array}{llll}
P(a_1) = \frac{40}{100} & P(a_2) = \frac{30}{100} & P(a_3) = \frac{10}{100} & P(a_4) = \frac{20}{100} \\
P(a_1|a_1) = \frac{15}{40} & P(a_2|a_1) = \frac{15}{40} & P(a_3|a_1) = \frac{4}{40} & P(a_4|a_1) = \frac{6}{40} \\
P(a_1|a_2) = \frac{13}{30} & P(a_2|a_2) = \frac{11}{30} & P(a_3|a_2) = \frac{1}{30} & P(a_4|a_2) = \frac{5}{30} \\
P(a_1|a_3) = \frac{5}{10} & P(a_2|a_3) = \frac{2}{10} & P(a_3|a_3) = \frac{2}{10} & P(a_4|a_3) = \frac{1}{10} \\
P(a_1|a_4) = \frac{7}{20} & P(a_2|a_4) = \frac{2}{20} & P(a_3|a_4) = \frac{3}{20} & P(a_4|a_4) = \frac{8}{20}
\end{array}
$$

Vaut-il mieux utiliser pour le compresser une méthode de Huffman d'ordre 1 (cf. le modèle de Markov d'ordre 1 de la page 44) ou l'algorithme de Huffman d'ordre 0 (celui présenté dans ce poly) ?

Exercice 3 Un expert en compression de données affirme détenir un logiciel de compression uniquement statistique permettant de compresser les deux textes suivants sans perte de données sur 38 bits :

- daaababababbccacccbdda
- abcdbabcdbccbbccabdb

Est-ce possible ? Justifiez votre réponse.

Exercice 4 Soit le texte «*abbcaaabcbabcbbdbadadadaccabb*». Calculez la redondance du texte stocké sous forme ASCII. On le compresse avec le code suivant :

$$a \equiv 0 \qquad b \equiv 10 \qquad c \equiv 111 \qquad d \equiv 110.$$

Quelle est la redondance avec ce code ?

Exercice 5 Soit X le texte de 2560 caractères créé en concaténant toutes les chaînes possibles composées de 10 fois le même caractère (X est générée grâce à l'algorithme suivant :

```
for (i=0; i<256; i++)
   for (j=0; j<10; j++)
      X[i*10+j] = (char) i;
```

autrement dit, cette chaîne est égale à "...aaaaaaaaaabbbbbbbbbbccccccccccddddddddddd..."). Quelle est, entre RLE et Huffman, la méthode qui compressera le mieux ce texte. Justifiez votre réponse. On supposera que RLE émet soit des triplets (flag, caractère, nombre de répétitions) dont les tailles sont respectivement 1 bit, 8 bits et 8 bits, soit des couples (flag, octet non compressé) dont les tailles sont respectivement 1 bit et 8 bits. C'est donc le bit de flag qui différencie une émission compressée d'une émission non compressée.

Exercice 6 Vous voulez compresser un texte à l'aide de Huffman. Vous avez effectué une première passe sur l'ensemble du texte, qui vous a permis de déterminer les nombres d'apparition des caractères suivants :

caractère	`a`	`c`	`d`	`e`	`l`	`n`	`o`	`q`	`r`	`s`	`t`	`u`	`v`	`z`	`'`	␣
nombre d'occurrences	70	8	22	80	33	23	2	3	34	55	9	3	7	7	2	20

Dessinez un arbre de Huffman correspondant puis compressez à l'aide de cet arbre le texte suivant : «`vous l'avez trouvee`». Dans la séquence de bits que vous écrirez, vous séparerez les caractères par des barres verticales. Par exemple, si un caractère `f` a pour code 0101 et si `g` a pour code 111, alors le code correspondant à «`fgf`» sera «0101|111|0101».

Si l'on ne tient pas compte du stockage de l'arbre dans le fichier résultat, quel est le taux de compression réalisé par Huffman sur la phrase que vous venez de compresser ?

Exercice 7 Calculez un code de Huffman pour le texte suivant : «bdadcdbdedbeecaeedbededee». Calculez la redondance ainsi que le taux de compression obtenus par ce code. Peut-on espérer diminuer la redondance en utilisant un autre arbre de Huffman ? En utilisant une passe RLE avant Huffman ?

Exercice 8 Dessinez les arbres successifs utilisés par l'algorithme de Huffman adaptatif pour encoder la phrase : «abcbdacEOF» et donnez le code résultant pour cette phrase.

Exercice 9 Soit le codage :

$$a \equiv 10 \qquad b \equiv 0 \qquad c \equiv 11.$$

Décodez la phrase «10011111000011100». Quelle est la redondance du code ci-dessus ? Son taux de compression ? Son facteur de compression ?
Réencodez la phrase avec l'algorithme de Huffman adaptatif. Doit-on s'attendre à un meilleur taux de compression ? Vérifiez en le calculant que votre réponse est bonne.

Exercice 10 Soit la phrase

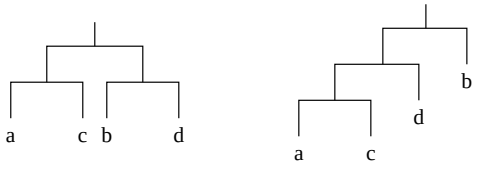
«ccdcabcccdaacdbaccacbcadcbaacccc».

Compressez-la en utilisant l'algorithme de Huffman. Parmi les algorithmes de compression statistique, pensez-vous qu'il existe un algorithme qui permette d'obtenir un taux de compression strictement meilleur ? Si oui, lequel ? Calculez la redondance obtenue avec le code de Huffman.

Exercice 11 Soit la phrase

«dcbcabbbacbddaabdbaaadabbacaabbcbcbaabdddbaddaabdd
bbcbaadabbccbddbbddbdbaddadabbbbbacaabddbbccbddbbd».

Lequel des 2 arbres de Huffman suivants est, d'après vous, le meilleur pour compresser cette phrase ? Justifiez.



arbre 1 arbre 2

Exercice 12 Soit un texte sur l'alphabet $\{a, b, \ldots, h, i\}$ vérifiant les probabilités :

$P(a) = \frac{1}{2}$ $P(b) = \frac{1}{4}$ $P(c) = \frac{1}{8}$ $P(d) = \frac{1}{16}$ $P(e) = \frac{1}{32}$ $P(f) = \frac{1}{64}$ $P(g) = \frac{1}{128}$ $P(h) = \frac{1}{256}$ $P(i) = \frac{1}{256}$.

Écrivez l'arbre de Huffman correspondant. Comparer avec l'arbre qui aurait été généré par un code de Golomb en posant $a \equiv 1$, $b \equiv 2$, $c \equiv 3$, etc.

Exercice 13 1/ Montrez que, quels que soient $p \geq 1$ et $q \geq p$,

$$\left(\sum_{k=p}^{q} \frac{1}{2^k}\right) + \frac{1}{2^q} = \frac{1}{2^{p-1}}.$$

En déduire que $\displaystyle\sum_{k=1}^{+\infty} \frac{1}{2^k} = 1$.

2/ Montrez que le code de Golomb est optimal pour la distribution $P(k) = \dfrac{1}{2^k}$ $\forall k \in \mathbb{N}$.

Exercice 14 Compressez à l'aide d'une méthode arithmétique, d'abord en précision infinie puis avec une précision de 4 décimales, le texte : «*abaaabac*EOF». Vous utiliserez pour cela les fréquences suivantes :

$$a \equiv 10\% \qquad b \equiv 30\% \qquad c \equiv 20\% \qquad d \equiv 30\% \qquad \text{EOF} \equiv 10\%.$$

Exercice 15 Décompressez à l'aide d'une méthode arithmétique le texte suivant : 0,04544372. Vous détaillerez les calculs que vous ferez, en particulier, vous écrirez les expressions mathématiques que vous calculerez. Vous pourrez effectuer les calculs avec une précision infinie.
Les intervalles sont décrits ci-dessous :

$$\text{a} : [0; 0,2[\qquad \text{b} : [0,2; 0,3[\qquad \text{c} : [0,3; 0,6[\qquad \text{d} : [0,6; 0,9[\qquad \text{EOF} : [0,9; 1[.$$

Exercice 16 Soit le texte suivant :

«eabbcaedacddbcaebdce»

1/ compressez ce texte à l'aide de l'algorithme de Huffman.
2/ quel est le taux de compression obtenu ?
3/ pour le texte ci-dessus, est-ce que Huffman est l'optimum de la classe des méthodes statistiques ? Justifiez votre réponse.

Exercice 17 Compressez grâce au codage arithmétique le texte «caabdaEOF». Vous supposerez pour cela que les intervalles des symboles sont respectivement :

$$\texttt{a} \equiv [0; 0,1[\qquad \texttt{b} \equiv [0,1; 0,5[\qquad \texttt{c} \equiv [0,5; 0,7[\qquad \texttt{d} \equiv [0,7; 0,9[\qquad \texttt{EOF} \equiv [0,9; 1[.$$

De plus, vous ferez vos calculs en entiers sur 4 digits (ainsi l'intervalle de départ sera $[0000; 9999[$).

Exercice 18
1/ Décompressez le code suivant, obtenu grâce à un codage arithmétique en nombres réels avec une précision infinie : 0,70271. Vous supposerez que, comme dans l'exercice précédent, les intervalles des symboles sont respectivement :

$$\texttt{a} \equiv [0; 0,1[\qquad \texttt{b} \equiv [0,1; 0,5[\qquad \texttt{c} \equiv [0,5; 0,7[\qquad \texttt{d} \equiv [0,7; 0,9[\qquad \texttt{EOF} \equiv [0,9; 1[.$$

2/ Compressez avec Huffman le texte obtenu en utilisant les mêmes fréquences que ci-dessus.
3/ En vous fondant sur les fréquences ci-dessus, calculez la redondance de Huffman pour ce texte.

3 Les méthodes à base de dictionnaires

Les méthodes de compression statistiques s'appuient sur un modèle statistique des données à compresser. Leur efficacité dépend fortement de l'adéquation du modèle avec les données. Les méthodes de compression à base de dictionnaires, elles, n'utilisent ni modèle statistique ni codes de tailles variables. Leur principe consiste à disposer d'un ensemble de chaînes de symboles (le dictionnaire) et à utiliser celui-ci pour encoder les chaînes de symboles formant le texte à compresser. Le dictionnaire peut être statique, il est alors créé une fois pour toutes, ou bien dynamique (on dit encore adaptatif) s'il est modifié par les chaînes rencontrées dans le texte à compresser.

Question : pourquoi développer des méthodes de compression à base de dictionnaire alors que les méthodes statistiques fonctionnent très bien ? À cela on peut fournir trois réponses :

1. Les opérations réalisées par les méthodes à base de dictionnaire sont des recherches de motifs et sont moins coûteuses en temps d'exécution que des opérations arithmétiques.
2. Les méthodes à base de dictionnaires sont asymmétriques, c'est-à-dire que l'encodeur et le décodeur n'effectuent pas les mêmes opérations. Il s'avère qu'ici le décodage est très simple, et par conséquent plus rapide que les méthodes statistiques. C'est un bon point si l'on ne compresse que rarement et que l'on décompresse souvent (fichiers compressés accessibles par internet par exemple).
3. En général, les méthodes de compression travaillant sur des chaînes de symboles sont plus efficaces que les méthodes travaillant sur les symboles indépendamment les uns des autres. Par exemple, prenons un alphabet à 2 symboles, a_1 et a_2, de probabilités P_1 et $1 - P_1$. L'entropie de l'alphabet est : $e(P_1) = H(P_1, 1 - P_1) = -P_1 \log_2 P_1 - (1 - P_1) \log_2(1 - P_1)$. On a alors

$$\frac{de(x)}{dx} = -\log_2 x + \log_2(1 - x).$$

Or cette fonction est positive sur $]0; 1/2]$ et négative sur $[1/2; 1[$. Par conséquent, comme $e(x)$ est une fonction symmétrique par rapport à $1/2$, l'entropie est minimale lorsque P_1 tends vers 0 ou vers 1. Cela signifie que les meilleurs taux de compression seront réalisés lorsque les probabilités des deux symboles sont très différentes, autrement dit, si X_1 est une variable aléatoire pouvant prendre pour valeurs les probabilités d'apparition des symboles, la compression est meilleure lorsque la variance de X_1 est grande. Or celle-ci a tendance à être plus grande sur des chaînes de symboles que sur des symboles isolés. Un exemple simple avec nos deux symboles. Supposons que $P_1 = 0,5$, c'est-à-dire qu'il y a autant de symboles a_1 que de symboles a_2. Alors la variance de X_1 est égale à 0 et l'entropie est à son maximum : 1. Supposons maintenant que les probabilités d'apparition des mots a_1a_1, a_1a_2 ,a_2a_1 et a_2a_2 soient respectivement 0,4, 0,1, 0,1 et 0,4. Alors on vérifie aisément que les probabilités d'apparition de a_1 et de a_2 sont bien de 0,5. De plus, si l'on appelle X_2 la variable aléatoire «probabilité d'apparition des différents mots de 2 lettres», alors

$$\mu(X_2) = 0,25 \qquad \text{et} \qquad V(X_2) = [2 \times (0,4 - 0,25)^2 + 2 \times (0,1 - 0,25)^2]/4 = 0,0225.$$

On voit donc bien que la variance est plus grande sur les chaînes de 2 symboles que sur les symboles individuels. L'entropie sera donc plus petite pour les mots de 2 symboles, et la compression meilleure.

Notons que ceci n'est vrai que d'une manière générale. En effet, il arrive que la compression soit meilleure lorsque l'on travaille sur des symboles isolément plutôt que sur des mots. Par exemple, si l'on compresse un texte composé de cinquante pourcents de a_1a_1 et de cinquante pourcents de a_1a_2. Alors la variance de X_2 est nulle tandis que celle de X_1 est égale à $[(0,75 - 0,5)^2 + (0,25 - 0,5)^2]/2 = 0,0625$.

Un exemple simple de compresseur à base de dictionnaire : prenez un dictionnaire de français. Numérotez chaque entrée dans celui-ci. Prenez maintenant un texte en français. À chaque mot, associez le numéro d'entrée du mot dans le dictionnaire. En fait, dans la plupart des textes, on s'aperçoit que très peu de mots différents sont utilisés. Il est bien rare d'avoir plus de 32768 mots de français différents. Par conséquent, si l'on crée un dictionnaire comportant les 32768 mots les plus courants de la langue française, chaque mot sera codé sur 2 octets, ce qui permet de réaliser une bonne compression du texte. Bien évidemment, si un mot du texte ne fait pas partie du dictionnaire, il faut prévoir un mécanisme pour l'écrire tel quel sur le flot de sortie (on peut utiliser un caractère d'échappement pour réaliser cela).

Le problème avec l'exemple ci-dessus est que si le texte à compresser est en anglais ou, pire encore, si c'est un fichier exécutable, on ne parviendra pas à réaliser une quelconque compression. Un dictionnaire statique n'est donc pas très efficace. Il vaudrait mieux avoir à sa disposition un dictionnaire qui évolue en fonction du texte en cours de compression. C'est ce que nous allons utiliser dans toute cette section.

L'idée est de partir d'un dictionnaire vide (ou d'un petit dictionnaire si l'on a des informations sur le type de fichier que l'on est en train de compresser). On lit alors les mots du flot d'entrée. Si ceux-ci font partie du dictionnaire, il suffit d'écrire sur le flot de sortie leurs index (numéros d'entrées), sinon, on les écrit tels quels sur le flot de sortie et on les rajoute dans le dictionnaire (éventuellement, on élimine d'anciens mots afin d'éviter que la taille du dictionnaire n'augmente trop).

Nous allons maintenant voir un certain nombre d'algorithmes, utilisés notamment par des logiciels comme `gzip`, `WinZip`, `PkZip`, `compress`, `ARC`, etc. Tous ces algorithmes sont des variations de deux algorithmes célèbres : LZ77 et LZ78, développés dans les années 70 par Jacob Ziv et Abraham Lempel, les deux fondateurs du domaine.

3.1 LZ77 ou le principe de la fenêtre coulissante

L'idée principale apportée par LZ77 est d'utiliser comme dictionnaire une partie du texte déjà compressé. L'encodeur maintient une petite fenêtre permettant de ne voir qu'une partie du flot d'entrée. À l'initialisation, un certain nombre de symboles de ce flot sont insérés dans la fenêtre de la gauche vers la droite. Lorsque de nouveaux symboles sont lus sur le flot d'entrée, ceux-ci sont insérés à droite dans la fenêtre et les symboles les plus à gauche sont éliminés. C'est donc bien le principe d'une fenêtre qui coulisse du début du texte à encoder vers la droite.

La fenêtre est divisée en deux comme le montre la figure ci-dessous :

début du texte...`les␣chaussettes␣de␣l'archidu`|`chesse␣sont␣seiches␣et␣archi`...reste du texte

La partie gauche, que les anglais appellent le *search buffer* et que nous appellerons tout bonnement la *fenêtre dictionnaire*, contient le dictionnaire des symboles les plus récents lus sur le flot d'entrée. La partie droite, appelée en anglais le *look-ahead buffer* et que nous appellerons la *fenêtre à compresser*, contient le texte non encore compressé. La partie à gauche de la fenêtre dictionnaire contient les plus vieux symboles déjà compressés. En pratique, la fenêtre dictionnaire a une longueur de quelques milliers d'octets tandis que la fenêtre à compresser n'est que de quelques dizaines d'octets.

Maintenant voyons comment l'algorithme fonctionne. Sur la figure ci-dessus, on voit que le premier caractère dans la fenêtre à compresser (celui le plus à gauche) est un «c». On va donc chercher un «c» dans la fenêtre dictionnaire en partant de la droite vers la gauche. On en trouve un en cinquième position («chidu»). On a donc trouvé dans le dictionnaire une chaîne de longueur 1 qui correspond exactement au début de la fenêtre à compresser. On va maintenant chercher s'il n'existe pas une chaîne identique plus grande. On regarde donc les caractères qui suivent les deux «c» : après le «c» de la fenêtre dictionnaire, il y a un «h», et c'est pareil dans la fenêtre à compresser. On a donc trouvé une chaîne de longueur 2 exactement identique au début de la fenêtre à compresser. On va maintenant chercher s'il existe une chaîne de longueur 3. Dans la fenêtre dictionnaire, «ch» est suivi par un «i», et dans la fenêtre à compresser «ch» est suivi par un «e». On doit donc rechercher la chaîne de longueur 3 plus à gauche dans la fenêtre dictionnaire. On retrouve à nouveau un «ch» en vingt quatrième position («chaussettes»). Malheureusement ce «ch» est suivi d'un «a» et non d'un «e». Il n'empêche que c'est une chaîne de longueur 2. Comme il n'y a pas d'autres «ch» à gauche, c'est cette entrée du dictionnaire que nous allons utiliser dans notre codage. Nous allons envoyer sur le flot de sortie le triplet (24,2,`e`). 24 représente l'index de la chaîne trouvée dans le dictionnaire, 2 correspond à sa longueur, et «`e`» est le prochain caractère dans la fenêtre à compresser.

> Dans LZ77, on sélectionne toujours dans le dictionnaire la plus longue chaîne de symboles identique à la chaîne en début de la fenêtre à compresser. Si le dictionnaire possède plusieurs chaînes de longueur maximale, on sélectionne la dernière (la plus à gauche).

En fait, on pourrait tout aussi bien sélectionner la première chaîne trouvée dans le dictionnaire, mais l'algorithme est alors légèrement plus complexe à écrire. On peut se dire que, de toutes façons, cela n'a aucune importance puisque les offsets (les déplacements pour trouver les chaînes de symboles dans le dictionnaire) sont codés sur un nombre fixe de bits. Cependant, sélectionner les chaînes les plus récentes peut présenter un avantage : si l'algorithme de compression utilise LZ77 en première étape, suivie par une méthode statistique,

les chaînes les plus récentes générant de petits offsets, il y a plus de chances que ceux-ci aient des codes plus petits en sortie de la méthode statistique. C'est ce qu'utilise LZH, une variante de LZ77 dûe à Bernd Herd.

Revenons au code que nous avons généré tout à l'heure : (24,2,e). Tous les codes générés par LZ77 sont des triplets :

> Les codes envoyés sur le flot de sortie par LZ77 sont des triplets (a, b, c), où :
> – a représente le déplacement que l'on doit effectuer à partir de la droite de la fenêtre dictionnaire pour pointer sur le début de la chaîne. Un déplacement de 1 pointe sur le caractère le plus à droite de cette fenêtre.
> – le b représente la longueur de la chaîne.
> – le c représente le caractère suivant la chaîne dans la fenêtre à compresser.
> Lorsqu'une chaîne a été codée, on fait glisser les deux fenêtres de $b + 1$ caractères vers la droite (b pour la longueur de la chaîne, et 1 pour le caractère c).
> Si, d'aventure, aucune chaîne n'est trouvée, un triplet $(0, 0, x)$ est émis, où x est le caractère le plus à gauche de la fenêtre à compresser. On fait alors glisser les fenêtres d'un seul caractère.

L'exemple de l'archiduchesse va nous aider à comprendre comment l'algorithme fonctionne :

Exemple 6 : Supposons donc que nous voulions encoder la phrase suivante :
`les␣chaussettes␣de␣l'archiduchesse␣sont␣seiches␣et␣archi␣seiches`. Les premières étapes de la compression sont alors :

fenêtre dictionnaire	fenêtre à compresser	
	`les␣chaussettes␣`	⇒ (0,0,l)
`l`	`es␣chaussettes␣d`	⇒ (0,0,e)
`le`	`s␣chaussettes␣de`	⇒ (0,0,s)
`les`	`␣chaussettes␣de␣`	⇒ (0,0,␣)
`les␣`	`chaussettes␣de␣l`	⇒ (0,0,c)
`les␣c`	`haussettes␣de␣l'`	⇒ (0,0,h)
`les␣ch`	`aussettes␣de␣l'a`	⇒ (0,0,a)
`les␣cha`	`ussettes␣de␣l'ar`	⇒ (0,0,u)
`les␣chau`	`ssettes␣de␣l'arc`	⇒ (6,1,s)
`les␣chaus`	`settes␣de␣l'arch`	⇒ (9,1,t)
`les␣chauss`	`ettes␣de␣l'archid`	⇒ (1,1,e)
`les␣chaussett`	`es␣de␣l'archiduch`	⇒ (12,2,d)
`les␣chaussettes␣d`	`e␣l'archiduchess`	⇒ (16,1,␣)
`les␣chaussettes␣de␣`	`l'archiduchesse␣`	⇒ (19,1,')
`es␣chaussettes␣de␣l'`	`archiduchesse␣so`	⇒ (15,1,r)
`␣chaussettes␣de␣l'archiduchesse␣`	`sont`	⇒ (19,2,i)
`aussettes␣de␣l'archiduchesse␣sont␣se`		⇒ (10,1,u)
`ssettes␣de␣l'archiduchesse␣sont␣seic`		⇒ (5,2,e)
`ttes␣de␣l'archiduchesse␣sont␣seiches`		⇒ (17,1,s)
`es␣de␣l'archiduchesse␣sont␣seiches␣e`		⇒ (16,2,s)
`de␣l'archiduchesse␣sont␣seiches␣et␣a`		⇒ (0,0,o)
`e␣l'archiduchesse␣sont␣seiches␣et␣ar`		⇒ (0,0,n)

On remarque qu'au début, il n'y a pas compression, il y a même expansion du texte (chaque fois qu'on a un triplet de la forme (0,0,x)). C'est seulement lorsque le dictionnaire commence à se remplir que la compression devient effective. ♦

Remarque : on a vu que LZ77 sélectionne les chaînes de caractères dans la fenêtre dictionnaire. Mais la seule limitation de l'algorithme est que le début de la chaîne appartienne à cette fenêtre. Rien n'empêche que la fin de la chaîne appartienne à la fenêtre à compresser. Par exemple, lorsque LZ77 rencontre la situation suivante :

`abcdefghijklmnoqrstuvwxyz`	`zzzzzzzzzabcdef`

alors LZ77 envoie sur le flot de sortie le triplet (1,9,a).

Le décodeur est beaucoup plus simple que l'encodeur. Il suffit en effet de maintenir un buffer de taille identique à celui de l'encodeur. Quand le décodeur lit un triplet, il lui suffit de rechercher la chaîne correspondante

dans sa fenêtre dictionnaire, d'écrire cette chaîne ainsi que le caractère en troisième position dans le triplet. Le décodeur fait alors coulisser la fenêtre comme l'aurait fait l'encodeur. Il peut alors lire un nouveau triplet, et ainsi de suite.

Notons qu'à première vue, contrairement aux méthodes statistiques, LZ77 n'a pas l'air de faire d'hypothèses sur les données du flot d'entrée[2]. Cependant, si l'on examine avec précision la méthode, on peut s'apercevoir qu'elle fait une hypothèse forte. En effet, de par la nature de la fenêtre coulissante, elle suppose que, dans le flot d'entrée, des motifs (chaînes de caractères) identiques sont toujours très proches les uns des autres. C'est en principe souvent vrai dans les textes en français. Pour preuve, le tableau ci-dessous montre comment LZ77 compresse le cours de probabilités (le programme utilisé est fourni dans la section 3.6.1). Ceci permet d'obtenir un taux de compression tout de même assez conséquent : 52%.

nom du fichier	taille du fichier d'origine		taille du fichier compressé par LZ77	
deust.tex	3809	octets	2027	octets
section1.tex	38142	octets	18299	octets
section2.tex	33569	octets	15383	octets
section3.tex	63915	octets	32179	octets
section4.tex	68099	octets	32391	octets
section5.tex	27463	octets	15483	octets
section6.tex	68757	octets	32675	octets
section7.tex	23096	octets	11631	octets
section8.tex	43803	octets	19843	octets
section9.tex	41920	octets	19011	octets
section10.tex	47436	octets	22539	octets
section11.tex	13442	octets	6639	octets
total	473451	octets	228100	octets

TAB. 13: Compression du cours de probabilité par l'algorithme LZ77

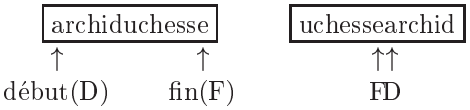
3.2 LZSS

Lorsqu'il a vu le jour[3], LZ77 a soulevé l'enthousiasme des chercheurs et des programmeurs. Assez rapidement, des variantes ont vu le jour, qui permettaient de réduire quelques défauts de la méthode. LZSS est une de ces variantes. Cette méthode offre trois améliorations par rapport à LZ77 :

1. elle utilise une queue circulaire pour stocker sa fenêtre à compresser ;
2. elle stocke sa fenêtre dictionnaire dans un arbre binaire de recherche ;
3. elle envoie sur le flot de sortie des couples plutôt que des triplets.

3.2.1 Une queue circulaire pour la fenêtre à compresser

Physiquement, une queue circulaire s'apparente à un tableau classique. La seule différence est qu'elle possède deux pointeurs : l'un indiquant le début du tableau et l'autre sa fin. Par exemple, les queues circulaires ci-dessous contiennent toutes les deux le mot «archiduchesse» :



L'intérêt pour LZ77 d'une telle structure est évident. Supposons que l'on ait dans notre fenêtre à compresser le texte suivant :

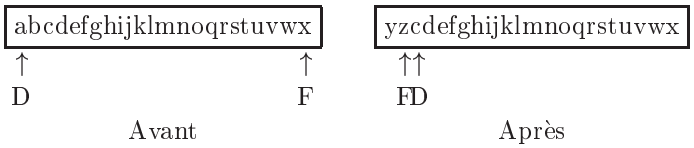
[2] L'hypothèse implicite faite par les méthodes statistiques était que les probabilités des symboles du flot d'entrée étaient différentes les unes des autres. Lorsque cette hypothèse n'était pas vérifiée, il n'y avait pas compression.

[3] cf. Jacob Ziv et Abraham Lempel (1977) «A universal algorithm for sequential Data compression», *IEEE Transactions on Information Theory* IT-23(3), pp.337–343.

et que l'on ait trouvé dans la fenêtre dictionnaire une chaîne de longueur 1. Dans ce cas il faut décaler la fenêtre ci-dessus de 2 crans vers la droite, puis insérer deux caractères, disons «yz». On obtient alors :

```
cdefghijklmnoqrstuvwxyz
```

Si l'on utilise un tableau classique, on a besoin de faire 22 décalages vers la gauche et 2 insertions. Si l'on utilise une queue circulaire, les deux insertions sont toujours nécessaires, mais on n'a plus qu'à décaler les pointeurs indiquant le début et la fin de la fenêtre :



Cette structure est évidemment très efficace pour gérer la fenêtre à compresser de LZ77. Elle le serait aussi pour gérer la fenêtre dictionnaire, mais LZSS utilise pour cela une autre structure : un arbre binaire de recherche.

3.2.2 Un arbre binaire de recherche pour la fenêtre dictionnaire

L'idée de LZ77 est de rechercher des chaînes de caractères dans la fenêtre dictionnaire. Or l'arbre binaire de recherche est la structure la plus indiquée pour effectuer des recherches. Rappelons qu'un arbre binaire de recherche est un arbre binaire dans lequel tous les noeuds du sous-arbre gauche de n'importe quel noeud A sont inférieurs (selon une certaine relation d'ordre) à A, et tous les noeuds du sous-arbre droit sont supérieurs à A.

Puisque LZ77 se contente de faire des recherches de chaînes de caractères, la relation d'ordre va donc permettre de comparer deux chaînes quelconques entre elles. La relation la plus simple est évidemment l'ordre lexicographique, c'est-à-dire l'ordre d'un dictionnaire de français classique. D'un point de vue informatique, on ne comparera évidemment pas des caractères a, b, c, etc, mais plutôt les nombres contenus dans chacun des octets des chaînes de caractères. Ainsi la chaîne hexadécimale «0x80 0x45» sera-t-elle plus grande que la chaîne «0x76 0x90».

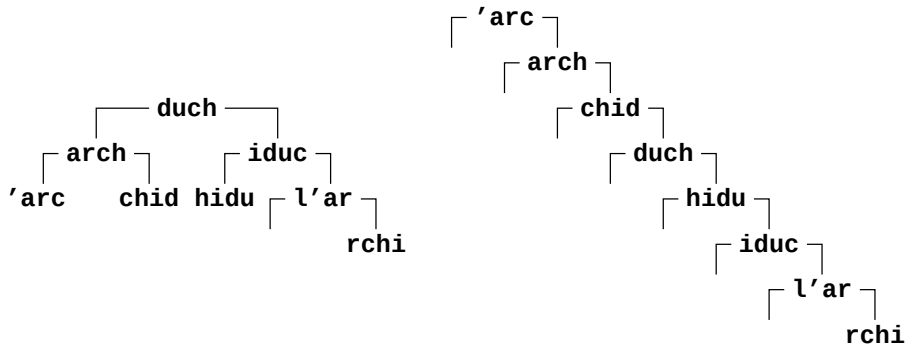
Voyons maintenant comment construire l'arbre, toujours sur l'exemple de l'archiduchesse. Supposons que nous ayons la fenêtre coulissante suivante :

```
l'archiduchesse
```

Dans ce cas, LZSS va générer tous les mots de 4 caractères que possède la fenêtre dictionnaire (4 parce que c'est le nombre de caractères de la fenêtre à compresser) :

mot	offset
`l'ar`	11
`'arc`	10
`arch`	09
`rchi`	08
`chid`	07
`hidu`	06
`iduc`	05
`duch`	04

Il suffit alors de ranger ces mots dans un arbre binaire de recherche. Évidemment, l'arbre n'est pas unique :



L'idéal est l'arbre bien balancé de gauche car c'est lui qui va permettre de limiter au maximum la recherche de la chaîne de caractères. Par exemple, dans la fenêtre à compresser, on a la chaîne «esse». On part de la racine de l'arbre, «esse» est plus grand que «duch», donc on va continuer la recherche sur le sous-arbre droit. «esse» est inférieur à «iduc», on continue sur le sous-arbre gauche, c'est encore inférieur à «hidu». En trois comparaisons de caractères, on a pu déterminer que LZ77 aurait dû émettre la séquence (0,0,e). Le pire des cas est l'arbre de droite, qui correspond exactement à la recherche effectuée dans LZ77.

Lorsque l'on décale la fenêtre dictionnaire dans LZ77, cela revient à éliminer les noeuds les plus anciens de l'arbre et à y rajouter de nouveaux noeuds. La taille de l'arbre est donc toujours la même. La mise à jour nécessite évidemment un peu de travail, mais globalement, comme les temps de recherche sont bien moins grands que dans LZ77, LZSS est un algorithme plus rapide que LZ77.

3.2.3 Émission sur le flot de sortie

La dernière amélioration de LZSS par rapport à LZ77 est qu'il n'émet pas des triplets comme dans LZ77. En principe, LZSS n'émet que des couples (déplacement, longueur de la chaîne). Si aucune chaîne n'a été trouvée, LZSS n'envoie pas un triplet (0,0,x) comme LZ77 mais seulement le caractère non compressé x. Pour distinguer les couples des caractères non compressés, il ajoute un bit (flag) supplémentaire aux couples et aux caractères non compressés.

En pratique, la fenêtre dictionnaire fait quelques milliers d'octets, ce qui nous donne des déplacements pouvant aller jusqu'à 11–13 bits. La taille du dictionnaire devrait être choisie de manière à ce que la taille totale d'un couple soit de 16 bits (sans compter le bit de flag). Cela permet de collecter des séquences de huit bits de flag dans un octet. Ainsi, la programmation de l'encodeur et du décodeur est facilitée : on n'a qu'à lire des octets sur le flot d'entrée. C'est tout de même beaucoup plus simple que les méthodes statistiques où l'on est amené à lire/écrire un nombre arbitraire de bits.

3.3 LZ78

LZ78 n'utilise ni fenêtre dictionnaire, ni fenêtre à compresser. En fait, il n'utilise pas du tout de fenêtre coulissante, mais plutôt un dictionnaire contenant toutes les chaînes de symboles rencontrées depuis le début de la compression sur le flot d'entrée. Au début, le dictionnaire est vide (ou pratiquement vide) et grossit au fur et à mesure que l'on avance dans la compression, sa seule limite étant la taille de la mémoire disponible.

L'encodeur émet des couples sur le flot de sortie. Le premier champ du couple correspond à un pointeur sur une entrée du dictionnaire, le second est le code d'un symbole (non compressé) du flot d'entrée. Au final, chaque couple correspond à une chaîne de symboles du flot d'entrée. Et c'est ce couple qui est rajouté au dictionnaire et qui est émis sur le flot de sortie. Dans LZ78, aucune entrée du dictionnaire n'est jamais détruite. Cela présente un avantage par rapport à LZ77 dans la mesure où les chaînes de symboles qui se répètent pouvaient être suffisamment éloignées pour ne pas faire partie de la même fenêtre coulissante. Dans ce cas, LZ77 ne pouvait pas compresser alors que LZ78 le peut. Par contre, cela présente un inconvénient majeur : le dictionnaire a tendance à occuper toute la mémoire disponible.

Voyons maintenant comment fonctionne précisément l'algorithme. Pour stocker le dictionnaire, nous allons utiliser un tableau (vecteur) de couples. Au début de l'algorithme, le tableau est vide, ce qui représente une chaîne vide de symboles. Lorsque des symboles sont lus sur le flot d'entrée, des chaînes sont rajoutées progressivement au dictionnaire en position 1, 2, 3, etc. Si le symbole a est lu sur le flot d'entrée, l'algorithme cherche dans le dictionnaire la chaîne «a», c'est-à-dire la chaîne composée exclusivement du symbole a. Si aucune chaîne n'est trouvée, on rajoute dans le dictionnaire le couple (0,a). 0 représente un pointeur vers une chaîne de symboles, en l'occurrence un 0 correspond à un pointeur sur la chaîne vide, l'\0 du langage C. (0,a) représente donc la concaténation de la chaîne vide avec la chaîne composée uniquement du symbole a, autrement dit, cela représente la chaîne «a». Supposons maintenant que le symbole a avait été trouvé en position 12 dans le dictionnaire, dans ce cas on lit le prochain symbole sur le flot d'entrée, appelons-le b. On cherche maintenant dans le dictionnaire la chaîne «ab». Si cette chaîne ne fait pas partie du dictionnaire, on la rajoute grâce au couple (12,b). (12,b) signifie la chaîne résultant de la concaténation de la chaîne se trouvant à l'indice 12 du tableau (dictionnaire) et de la chaîne «b». Puisqu'à l'indice 12, on avait trouvé la chaîne «a», (12,b) représente bien la chaîne «ab». Si «ab» avait été trouvée dans le dictionnaire on continuerait avec le prochain symbole du flot d'entrée, appelons-le c,

et on rechercherait de la même manière la chaîne «*abc*», et ainsi de suite jusqu'à ce qu'on ait lu la totalité du flot d'entrée.

En général, les premiers symboles lus sur le flot d'entrée deviennent des chaînes à 1 symbole dans le dictionnaire. Il n'y a alors pas de compression de réalisée. Par contre, au fur et à mesure qu'on lit des symboles sur le flot d'entrée, les chaînes de symboles contenues dans le dictionnaire deviennent de plus en plus longues et on commence à avoir de la compression. Voyons maintenant un exemple illustrant le fonctionnement de l'algorithme :

Exemple 7 : On va compresser la chaîne :

«les␣chaussettes␣de␣l'archiduchesse␣sont␣seiches␣et␣archi␣seiches.».

Tout d'abord, on va partir d'un tableau (dictionnaire) vide, et on va lire le premier symbole du flot d'entrée, ici un «l». On recherche «l» dans le dictionnaire. Évidemment, on ne le trouve pas. On va donc ajouter un couple (0,l) au tableau et émettre ce couple sur le flot de sortie. On obtient donc le dictionnaire suivant :

indices	couples	chaînes correspondantes
0	null	
1	(0,l)	l

Le même couple est émis sur le flot de sortie. On lit maintenant sur le flot d'entrée le caractère «e». Là encore, on le recherche dans le dictionnaire mais on ne le trouve pas. On rajoute donc au dictionnaire le couple (0,e). Idem pour les caractères, s, ␣, c, h, a et u. Après avoir lu ces caractères on obtient le dictionnaire suivant :

indices	couples	chaînes correspondantes
0	null	
1	(0,l)	l
2	(0,e)	e
3	(0,s)	s
4	(0,␣)	␣
5	(0,c)	c
6	(0,h)	h
7	(0,a)	a
8	(0,u)	u

On lit maintenant un «s» sur le flot d'entrée. On le recherche dans le dictionnaire et on le trouve à l'indice 3. On va donc lire le prochain caractère sur le flot d'entrée. C'est encore un «s». On recherche donc maintenant dans le dictionnaire la chaîne «ss». On ne la trouve pas, donc on va la rajouter au dictionnaire. Comment ? Simplement en concaténant la chaîne «s» qu'on avait trouvé à l'indice 3 avec la chaîne «s» provenant du dernier caractère lu sur le flot d'entrée, ce qui revient à rajouter le couple (3,s) dans le dictionnaire :

indices	couples	chaînes correspondantes
0	null	
1	(0,l)	l
2	(0,e)	e
3	(0,s)	s
4	(0,␣)	␣
5	(0,c)	c
6	(0,h)	h
7	(0,a)	a
8	(0,u)	u
9	(3,s)	ss

On lit maintenant un «e» sur le flot d'entrée. On le recherche dans le dictionnaire et on le trouve à l'indice 2. On lit le prochain caractère sur le flot d'entrée, c'est un «t». On recherche donc maintenant la chaîne «et» dans le dictionnaire. On ne la trouve pas, donc on la rajoute dans le dictionnaire, i.e., on rajoute le couple (2,t) :

indices	couples	chaînes correspondantes
00	null	
01	(0,l)	l
02	(0,e)	e
03	(0,s)	s
04	(0,␣)	␣
05	(0,c)	c
06	(0,h)	h
07	(0,a)	a
08	(0,u)	u
09	(3,s)	ss
10	(2,t)	et

On continue ainsi avec les caractères de la chaîne «tes␣de␣l'archiduche», ce qui nous donne le dictionnaire suivant :

indices	couples	chaînes correspondantes
00	null	
01	(00,l)	l
02	(00,e)	e
03	(00,s)	s
04	(00,␣)	␣
05	(00,c)	c
06	(00,h)	h
07	(00,a)	a
08	(00,u)	u
09	(03,s)	ss
10	(02,t)	et

indices	couples	chaînes correspondantes
11	(00,t)	t
12	(02,s)	es
13	(04,d)	␣d
14	(02,␣)	e␣
15	(01,')	l'
16	(07,r)	ar
17	(05,h)	ch
18	(00,i)	i
19	(00,d)	d
20	(08,c)	uc
21	(06,e)	he

On lit maintenant le caractère «s» sur le flot d'entrée. On le trouve à l'indice 3 du tableau. On lit le prochain caractère sur le flot d'entrée, i.e., à nouveau un «s». On recherche donc maintenant la chaîne «ss» dans le dictionnaire. On la trouve à l'indice 9. On lit le prochain caractère sur le flot d'entrée, c'est un «e». Là encore, on recherche la chaîne «sse» dans le dictionnaire. On ne la trouve pas, on va donc la rajouter grâce au couple (9,e). Et ainsi de suite. Quand on a lu tout le flot d'entrée on obtient le dictionnaire :

indices	couples	chaînes correspondantes
00	null	
01	(00,l)	l
02	(00,e)	e
03	(00,s)	s
04	(00,␣)	␣
05	(00,c)	c
06	(00,h)	h
07	(00,a)	a
08	(00,u)	u
09	(03,s)	ss
10	(02,t)	et
11	(00,t)	t
12	(02,s)	es
13	(04,d)	␣d
14	(02,␣)	e␣
15	(01,')	l'
16	(07,r)	ar
17	(05,h)	ch
18	(00,i)	i

indices	couples	chaînes correspondantes
19	(00,d)	d
20	(08,c)	uc
21	(06,e)	he
22	(09,e)	sse
23	(04,s)	␣s
24	(00,o)	o
25	(00,n)	n
26	(11,␣)	t␣
27	(03,e)	se
28	(18,c)	ic
29	(21,s)	hes
30	(04,e)	␣e
31	(26,a)	t␣a
32	(00,r)	r
33	(17,i)	chi
34	(23,e)	␣se
35	(28,h)	ich
36	(12,.)	es.

Notons qu'à chaque fois que l'on a rajouté un couple au dictionnaire, le même couple était envoyé sur le flot de sortie. Autrement dit, on a envoyé sur ce dernier la séquence : (0,l) (0,e) (0,s) (0,␣) (0,c) (0,h) (0,a) (0,u) (3,s) (2,t) (0,t) (2,s) (4,d) (2,␣) (1,') (7,r) (5,h) (0,i) (0,d) (8,c) (6,e) (9,e) (4,s) (0,o) (0,n) (11,␣) (3,e) (18,c) (21,s) (4,e) (26,a) (0,r) (17,i) (23,e) (28,h) (12,.). ♦

Le principe de décodage est similaire à celui du codage : on lit un couple sur le flot d'entrée. On écrit sur le flot de sortie la chaîne pointée par le premier élément du couple, puis le caractère correspondant au deuxième élément du couple. Enfin on rajoute ce couple au dictionnaire. On recommence alors ce processus avec un autre couple du flot d'entrée.

En pratique, si l'on veut effectuer des recherches rapidement dans le dictionnaire, la structure en tableau n'est pas adaptée. Il vaut mieux utiliser une structure d'arbre : sur la figure ci-après, le numéro avant le underscore correspond à l'indice dans le tableau ci-avant. Cet underscore est suivi du couple stocké dans le tableau. L'arbre est construit de telle sorte que le parent de chaque noeud corresponde au premier élément du couple. Bien évidemment, en pratique, seuls des pointeurs seront utilisés et on ne stockera pas un couple, mais seulement le deuxième élément du couple. Un simple aperçu de l'arbre montre que la recherche est bien plus rapide avec cette structure qu'avec un tableau.

Ainsi, l'ajout d'une phrase dans le dictionnaire correspond à l'ajout d'une branche dans l'arbre. LZ78, rappelons-le, ne considère que des ajouts et jamais de suppression. Bien évidemment, comme la taille de la mémoire disponible n'est pas infinie, celle-ci risque d'être complètement saturée par l'algorithme. En fait, elle le devient assez rapidement sauf quand on compresse de tout petits fichiers. Il faut donc trouver un moyen de limiter cette consommation excessive de mémoire. Comme LZ78 ne précise pas comment le faire, différentes techniques ont été utilisées dans les logiciels du commerce :

- La plus simple des solutions est de geler le dictionnaire à partir d'un certain point. Avant, le dictionnaire est donc dynamique, ou encore adaptatif, et après ce point, il devient statique. C'est la méthode qu'utilise en partie `compress`, l'un des logiciels que l'on trouve sous UNIX/Linux.
- On peut détruire tout le dictionnaire lorsque celui-ci a été rempli au maximum et en reconstruire un nouveau en repartant d'un arbre vide. Cette solution revient à séparer le fichier d'entrée en blocs, chaque bloc ayant son propre dictionnaire. C'est une bonne solution si les données du flot d'entrée varient beaucoup d'un bloc à l'autre. Par contre, si elles sont assez liées, on perd beaucoup en efficacité. Un autre avantage de la méthode : elle est moins sensible aux erreurs de transmission. En effet, supposons que l'on ait une grosse archive compressée par bloc avec LZ78. L'archive est stockée sur disquette, mais, malheureusement, lorsqu'on lit l'archive, on s'aperçoit que le troisième secteur de la disquette sur laquelle elle était stockée est défectueux. Dans ce cas, tout le dictionnaire risque d'être altéré et donc l'archive risque de ne pas pouvoir être décompressée du tout. Si l'on divise le flot d'entrée par blocs, seul le dictionnaire d'un bloc aura été altéré et on pourra récupérer tout le reste de l'archive.
- La dernière méthode, certainement la plus efficace en terme de taux de compression mais aussi la plus délicate à réaliser, consiste à éliminer les phrases les plus anciennes du dictionnaire. À ce jour, il n'existe pas de bon algorithme pour réaliser cette opération.

Dans l'algorithme prsenté dans la section 3.6.2, j'ai choisi de vider le dictionnaire à chaque fois que celui-ci était plein. Bien que ce ne soit clairement pas la méthode la plus efficace, elle permet d'obtenir des résultats tout à fait acceptables, comme le montre le tableau suivant (taux de compression global 36%) :

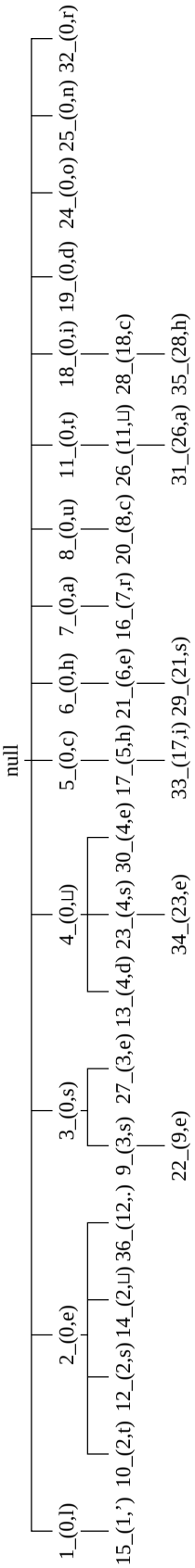


FIG. 9 – L'arbre-dictionnaire de l'exemple ci-dessus

nom du fichier	taille du fichier d'origine		taille du fichier compressé par LZ78	
deust.tex	3809	octets	2861	octets
section1.tex	38142	octets	25253	octets
section2.tex	33569	octets	22493	octets
section3.tex	63915	octets	40478	octets
section4.tex	68099	octets	42380	octets
section5.tex	27463	octets	19163	octets
section6.tex	68757	octets	42677	octets
section7.tex	23096	octets	16277	octets
section8.tex	43803	octets	27773	octets
section9.tex	41920	octets	27128	octets
section10.tex	47436	octets	30920	octets
section11.tex	13442	octets	7994	octets
total	473451	octets	305397	octets

TAB. 14: Compression du cours de probabilité par l'algorithme LZ78

3.4 LZW

LZ78 a suscité beaucoup d'engoûment, et de nombreuses variantes ont vu le jour. L'une des plus populaires est certainement LZW. Celle-ci a été développée en 1984 par T.A. Welch. Sa caractéristique principale est d'éliminer le second champ dans les couples LZ78. LZW n'émet donc que des pointeurs sur des phrases et jamais de caractère. Étrange non ? Afin de bien comprendre comment fonctionne la méthode, nous allons supposer que le dictionnaire est stocké sous la forme d'un tableau, comme au début de LZ78. Nous verrons par la suite comment est effectivement implémenté le dictionnaire dans LZW.

3.4.1 Encodage LZW

Sans plus attendre, arrêtons cet insoutenable suspense et voyons le principe de LZW : au début de l'algorithme, on initialise le dictionnaire avec l'ensemble de toutes les phrases de 1 symbole. Si l'on travaille sur des caractères, comme on le fait d'habitude, on initialise les 256 premières entrées du dictionnaire avec tous les caractères possibles. L'encodeur lit maintenant sur le flot d'entrée les symboles un à un et les concatène pour former une chaîne de symboles que nous appellerons C. Après l'ajout de chaque symbole dans C, LZW recherche dans le dictionnaire s'il contient cette nouvelle chaîne. Tant qu'elle appartient au dictionnaire, on continue à lire des symboles sur le flot d'entrée et à les concaténer à C. Cependant, à un certain point, l'ajout du symbole x à C va faire que la chaîne Cx ne va pas appartenir au dictionnaire. Dans ce cas, C appartient au dictionnaire, mais pas Cx. L'encodeur émet alors sur le flot de sortie l'index qui pointe sur la chaîne C dans le dictionnaire et il sauve la chaîne Cx à la fin du dictionnaire. Enfin, l'encodeur réinitialise la chaîne C à «x». Illustrons l'algorithme sur l'exemple de l'archiduchesse :

Exemple 8 : On veut donc compresser avec LZW la chaîne :

`«les␣chaussettes␣de␣l'archiduchesse␣sont␣seiches␣et␣archi␣seiches».`

À l'initialisation de l'algorithme, les 256 premières entrées du dictionnaire sont remplies avec l'ensemble de tous les caractères possibles :

index	chaîne	index	chaîne	index	chaîne
000	NUL '\0'	102	f	116	t
001	SOH	103	g	117	u
002	STX	104	h	118	v
003	ETX	105	i	119	w
004	EOT	106	j	120	x
⋮	⋮	107	k	121	y
065	A	108	l	122	z
066	B	109	m	123	{
⋮	⋮	110	n	124	|
097	a	111	o	125	}
098	b	112	p	126	~
099	c	113	q	127	DEL
100	d	114	r	⋮	⋮
101	e	115	s	255	255

La chaîne C est initialisée à la chaîne vide. On peut maintenant lire le flot d'entrée. Le premier symbole qu'on lit est un «`l`». La nouvelle chaîne C est donc : Cl = «l». On recherche dans le dictionnaire s'il existe une entrée correspondant à cette chaîne. On la trouve à l'indice 108. Puisqu'on l'a trouvée, on lit le prochain symbole sur le flot d'entrée : un «`e`». On recherche donc maintenant si la chaîne «le» appartient au dictionnaire. Ce n'est pas le cas, donc on la rajoute à la première place disponible, c'est-à-dire à l'indice 256 du tableau. Le dictionnaire devient donc :

index	chaîne	index	chaîne	index	chaîne
000	NUL '\0'	103	g	117	u
001	SOH	104	h	118	v
002	STX	105	i	119	w
003	ETX	106	j	120	x
⋮	⋮	107	k	121	y
065	A	108	l	122	z
066	B	109	m	123	{
⋮	⋮	110	n	124	|
097	a	111	o	125	}
098	b	112	p	126	~
099	c	113	q	127	DEL
100	d	114	r	⋮	⋮
101	e	115	s	255	255
102	f	116	t	256	le

On émet sur le flot de sortie la valeur 108, c'est-à-dire le pointeur sur le dernier C trouvé dans le dictionnaire avant l'ajout de «`le`». Lorsque 108 a été émis, on réinitialise C à la valeur «`e`», c'est-à-dire le symbole qui avait fait échouer la recherche dans le dictionnaire.

Avec C = «`e`», on recommence le processus. Dans le dictionnaire, cette chaîne a pour indice 101. On relit un caractère sur le flot d'entrée : c'est un «`s`». La chaîne «`es`» n'appartient pas au dictionnaire. On la rajoute donc à l'indice 257 du tableau et on émet le nombre 101 sur le flot de sortie. La chaîne C est alors réinitialisée à «`s`», qui a pour indice 115. Le prochain symbole est un ␣, un espace. La chaîne «`s`␣» n'appartient pas au dictionnaire, on la rajoute donc à l'indice 258 du tableau et on émet 115 sur le flot de sortie. Ainsi, après avoir lu sur le flot d'entrée la chaîne «`les␣chaussette`», on obtient le dictionnaire suivant :

index	chaîne	index	chaîne	index	chaîne
000	NUL '\0'	110	n	125	}
⋮	⋮	111	o	⋮	⋮
097	a	112	p	255	255
098	b	113	q	256	`le`
099	c	114	r	257	`es`
100	d	115	s	258	`s␣`
101	e	116	t	259	`␣c`
102	f	117	u	260	`ch`
103	g	118	v	261	`ha`
104	h	119	w	262	`au`
105	i	120	x	263	`us`
106	j	121	y	264	`ss`
107	k	122	z	265	`se`
108	l	123	{	266	`et`
109	m	124	|	267	`te`

De plus, on a déjà émis sur le flot de sortie la séquence : 108 101 115 32 99 104 097 117 115 115 101 116. On va maintenant lire le «`s`» à la fin de «`chaussettes`». C contient la chaîne «`e`». On recherche donc maintenant la chaîne «`es`», que l'on trouve à l'indice 257. On lit le caractère suivant sur le flot d'entrée, c'est un ␣. La chaîne «`es␣`» n'appartient pas au dictionnaire, on la rajoute donc à l'indice 268, et on émet un 257 sur le flot de sortie. C est de plus réinitialisé à «␣» dont l'indice est 32. Et ainsi de suite. À la fin on obtient le dictionnaire suivant :

index	chaîne	index	chaîne	index	chaîne	index	chaîne	index	chaîne
000	NUL '\0'	111	o	255	255	271	`e␣`	287	`nt`
⋮	⋮	112	p	256	`le`	272	`␣l`	288	`t␣`
097	a	113	q	257	`es`	273	`l'`	289	`␣se`
098	b	114	r	258	`s␣`	274	`'a`	290	`ei`
099	c	115	s	259	`␣c`	275	`ar`	291	`ic`
100	d	116	t	260	`ch`	276	`rc`	292	`ches`
101	e	117	u	261	`ha`	277	`chi`	293	`s␣e`
102	f	118	v	262	`au`	278	`id`	294	`et␣`
103	g	119	w	263	`us`	279	`du`	295	`␣a`
104	h	120	x	264	`ss`	280	`uc`	296	`arc`
105	i	121	y	265	`se`	281	`che`	297	`chi␣`
106	j	122	z	266	`et`	282	`ess`	298	`␣sei`
107	k	123	{	267	`te`	283	`se␣`	299	`ich`
108	l	124	|	268	`es␣`	284	`␣s`		
109	m	125	}	269	`␣d`	285	`so`		
110	n	⋮	⋮	270	`de`	286	`on`		

À la fin, on aura émis en sortie la séquence : 108 101 115 32 99 104 097 117 115 115 101 116 257 32 100 101 32 108 39 97 114 260 105 100 117 260 257 265 32 115 111 110 116 284 101 105 281 258 266 32 275 277 289 291 292 EOF. ♦

3.4.2 Décodage LZW

Afin de bien comprendre comment fonctionne le décodeur, rappelons les étapes du codage : tout d'abord, les 256 premières entrées du dictionnaire sont initialisées aux valeurs de tous les octets possibles. Ensuite, les opérations suivantes sont effectuées :

1. on lit un symbole sur le flot d'entrée, appelons-le x
2. on recherche si la chaîne Cx appartient au dictionnaire. Le cas échéant, on rajoute x à C, i.e., $C := Cx$, et on revient à l'étape 1. Sinon on passe à l'étape 3.

3. On rajoute dans le dictionnaire la chaîne Cx. On émet sur le flot de sortie l'indice qui pointe sur C dans le dictionnaire. On réinitialise C à x, i.e., $C := x$. On revient à l'étape 1.

Le décodeur commence de la même manière que l'encodeur : il remplit les 256 premières entrées de son dictionnaire avec tous les octets possibles. Il lit alors les pointeurs sur le flot d'entrée et les utilise à la fois pour augmenter son dictionnaire (de la même manière que l'encodeur) et pour émettre des symboles sur son flot de sortie. Il commence par lire le premier pointeur sur son flot d'entrée. Celui-ci lui fournit l'indice d'une chaîne C. On peut déjà émettre cette chaîne sur le flot de sortie. Pour bien faire, il faudrait aussi rajouter une chaîne du style Cx au dictionnaire de l'encodeur. Malheureusement, on ne connaît pas pour l'instant le x qu'il faudrait rajouter. On va donc sauvegarder la valeur de C pour le moment, disons dans une variable D, et continuer l'algorithme. On lit maintenant le deuxième pointeur sur le flot d'entrée, ce qui nous fournit une nouvelle chaîne C. Hola ! Stop ! rappelez-vous comment fonctionne l'encodeur, on est sûr que la première lettre de la nouvelle chaîne C correspond exactement au x qu'on attendait. On rajoute donc au dictionnaire la chaîne Dx. On n'oublie pas d'émettre la chaîne C sur le flot de sortie. On sauvegarde alors $C : D := C$, et on recommence ainsi de suite jusqu'à ce qu'on ait lu tout le flot d'entrée.

Exemple 8 (suite) : Nous avions vu dans la sous-section précédente que la chaîne de caractères :

`«les␣chaussettes␣de␣l'archiduchesse␣sont␣seiches␣et␣archi␣seiches»`

était compressée de la manière suivante par LZW : 108 101 115 32 99 104 097 117 115 115 101 116 257 32 100 101 32 108 39 97 114 260 105 100 117 260 257 265 32 115 111 110 116 284 101 105 281 258 266 32 275 277 289 291 292 EOF. Notre mission, que nous acceptons, consiste maintenant à décoder ce message. Au début de l'algorithme, nous construisons un dictionnaire qui contient les 256 octets :

index	chaîne	index	chaîne	index	chaîne
000	NUL '\0'	102	f	116	t
001	SOH	103	g	117	u
002	STX	104	h	118	v
003	ETX	105	i	119	w
004	EOT	106	j	120	x
⋮	⋮	107	k	121	y
065	A	108	l	122	z
066	B	109	m	123	{
⋮	⋮	110	n	124	|
097	a	111	o	125	}
098	b	112	p	126	~
099	c	113	q	127	DEL
100	d	114	r	⋮	⋮
101	e	115	s	255	255

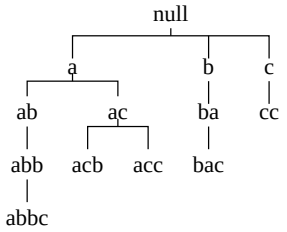
On peut maintenant lire le premier indice sur le flot d'entrée : 108. Dans notre dictionnaire, 108 correspond à un `«l»`. On émet donc ce caractère sur le flot de sortie et on sauvegarde `«l»` dans D. Deuxième pointeur : 101, qui correspond à la chaîne $C =$ `«e»` dans le dictionnaire. Le premier caractère de cette chaîne est un `«e»`, donc $Dx =$ `«le»`. C'est cette chaîne que l'on doit sauvegarder dans le dictionnaire, à la première place disponible, c'est-à-dire 256. Remarquons que cela correspond exactement au dictionnaire qu'on avait créé lors de l'encodage. On émet la chaîne C, i.e., `«e»`, sur le flot de sortie. On sauvegarde cette chaîne dans D : $D =$ `«e»`. Le prochain indice est 115, ce qui correspond à $C =$ `«s»`. On rajoute donc à D le premier caractère de C, à savoir `«s»`, ce qui nous donne $D =$ `«es»`, et on sauve cette chaîne dans le dictionnaire. Là encore, on remarque que l'on reconstitue exactement le dictionnaire que l'on avait construit lors de l'encodage. On émet C sur le flot de sortie et on le sauvegarde dans D : $D = C$. Et ainsi de suite. Remarquons que les chaînes émises sur le flot de sortie forment la chaîne `«les»`, qui correspond bien au début de la phrase compressée. ♦

3.4.3 Stockage du dictionnaire

Jusqu'à maintenant, nous avons supposé que le dictionnaire était en fait un tableau, qui grossissait au fur et à mesure que l'on avançait dans l'algorithme. On voit bien que, comme dans LZ78, il est beaucoup plus intéressant d'utiliser une structure d'arbre. D'abord parce que les recherches en sont facilitées et sont plus rapides. Ensuite parce que, si l'on utilise une structure d'arbre, on n'a à stocker dans les noeuds qu'un seul caractère, alors que la structure en tableau nécessite de stocker des phrases de longueurs variables.

Une structure d'arbre est donc tout à fait indiquée pour stocker le dictionnaire. Les noeuds de cet arbre peuvent avoir de 0 à 256 fils. Afin de ne pas saturer la mémoire, il faut donc prévoir un mécanisme permettant de n'avoir pas à réserver automatiquement de la place pour ces 256 fils potentiels. Une méthode possible serait de stocker les fils dans une liste chaînée. Mais les pointeurs nécessaires au chaînage prennent beaucoup trop de place. Il faudrait donc trouver une autre structure. Celle-ci existe et s'appelle une *table de hachage*.

L'idée est d'utiliser un tableau pour stocker l'arbre. Chaque élément de ce tableau contient deux champs : (1) le symbole qu'on aurait stocké dans le noeud de l'arbre et (2) un pointeur vers son parent dans l'arbre. Supposons par exemple que nous ayons l'arbre suivant :



Alors, on peut sauvegarder cet arbre dans le tableau suivant :

index	pointeur sur parent	symbole	chaîne correspondante
000	000	null	null
⋮	⋮	⋮	
097	000	a	a
098	000	b	b
099	000	c	c
⋮	⋮	⋮	
255	000	255	255
256	097	b	ab
257	098	a	ba
⋮	⋮	⋮	
270	257	a	bac
271	099	c	cc
272	097	c	ac
273	256	b	abb
274	272	b	acb
275	273	c	abbc
276	272	c	acc

Pour remonter dans l'arbre, il suffit d'utiliser le champ «pointeur sur parent». Pour descendre dans l'arbre d'un noeud vers son fils, on utilise une fonction de hachage : le pointeur du noeud et le symbole du fils sont hachés pour obtenir l'index correspondant au fils. Supposons par exemple que la chaîne «abb» appartienne déjà au dictionnaire, le «a» étant stocké à l'index 97, la chaîne «ab» à l'index 256 et la chaîne «abb» à l'index 273. On veut accéder à la chaîne «abbc» et on se trouve pour l'instant à la racine de l'arbre. On hache le pointeur sur `null` avec la lettre «a», on obtient alors l'index correspondant à la chaîne «a», c'est-à-dire 97. On va maintenant hacher 97 et «b» afin d'obtenir l'index correspondant à chaîne «ab». On obtient alors 256. On recommence : on hache 256 avec «b» pour obtenir l'index correspondant à «abb». On obtient alors le numéro 273. Et ainsi de suite.

Supposons maintenant que la chaîne «abbc» n'appartienne pas au dictionnaire et que nous voulions la

rajouter. Le paragraphe précédent nous a permis d'accéder à son noeud père, c'est-à-dire «abb». Afin d'obtenir l'index correspondant à «abbc», on va hacher 273 et «c». On obtient par exemple le numéro 275. Plusieurs possibilités s'offrent à nous :

1. 275 est l'index d'une case non encore utilisée du tableau. Dans ce cas, pas de problème, on met à jour cette case avec les informations : pointeur sur parent = 273 et symbole = «c».
2. 275 contient déjà un noeud dont le parent est 273 et dont le symbole est «c». Dans ce cas, on n'a rien à faire.
3. 275 contient un autre noeud. Cela peut arriver si le hachage d'un autre pointeur avec un autre symbole avait déjà eu pour résultat l'index 275. C'est ce qu'on appelle une *collision*. Dans ce cas, on va insérer le couple (273, «c») dans la première case suivante qui est inutilisée. Si toutes les cases suivantes sont utilisées, on revient au début du tableau et on le parcourt jusqu'à obtenir une case vide.

En pratique, le problème des collisions nous oblige à avoir trois champs dans les cases du tableau :

1. le pointeur sur le parent ;
2. l'index obtenu par la fonction de hachage ;
3. le symbole du noeud.

Le deuxième champ est nécessaire pour être certain d'accéder au bon élément du tableau. En effet, supposons que l'on ait le tableau suivant :

index	pointeur sur parent	symbole	chaîne correspondante
263	097	`b`	`ab`
264	256	`d`	`cd`
265	263	`e`	`abe`

On veut rajouter la chaîne «abd». La fonction de hachage nous fournit l'index 264. Malheureusement, cette case est déjà occupée et la première case vide se trouve en 266. On écrit donc «abd» en 266. Si on recherche cette chaîne par la suite, la fonction de hachage nous donnera l'index 264. Il ne suffira alors pas de rechercher à partir de 264 les cases dont le pointeur sur parent est 263 pour trouver la bonne chaîne. De même, il ne suffit pas de rechercher les cases dont le symbole est un «d». Si l'on omet le deuxième champ, on doit alors faire beaucoup trop de comparaisons lors des recherches et cela nuit gravement à la rapidité d'exécution du compresseur. Illustrons le procédé de recherche/insertion sur un petit exemple :

Exemple 9 : Supposons que l'on veuille compresser la chaîne «ababab...». Nous allons utiliser la structure de dictionnaire suivante :

```
typedef struct {
  unsigned int parent;
  unsigned int index;
  unsigned char symbole;
  } dico;
```

Au début de la compression, on s'alloue un tableau de type `dico` de taille ce que l'on veut. Appelons `dict` ce tableau. Normalement, on devrait allouer les cases correspondant aux 256 octets possibles. Ici, pour simplifier un peu, comme on n'a que deux lettres dans l'alphabet, `a` et `b`, on n'initialisera que deux cases, et on supposera que la fonction de hachage pour ces deux lettres donnera respectivement les indices 1 et 2 du tableau. Toutes les autres cases du tableau seront marquées comme non utilisées. On part donc d'un tableau de la forme :

parent	0	0	/	/	/	···
index	1	2	-	-	-	···
symbole	`a`	`b`				···

L'initialisation de l'algorithme est terminée. On peut maintenant commencer à lire les symboles sur le flot d'entrée. Le premier de ces messieurs est un «a». On le stocke dans une variable, appelons-la C. En fait, ce qui est réellement stocké, ce n'est pas «a» mais la valeur de son index, c'est-à-dire 1. Par conséquent, $C = 1$. Puisque c'est le premier symbole, l'encodeur supposera qu'il appartient au dictionnaire et donc ne le recherchera pas.

Passons maintenant au deuxième symbole. C'est un «b». Stockons l'index de ce symbole dans une variable D. D'après notre dictionnaire, $D = 2$. L'encodeur doit maintenant rechercher si la chaîne «ab» appartient au dictionnaire. Pour cela, il va calculer `pointeur := hash(C,D)`, c'est-à-dire qu'il calcule grâce à la fonction de hachage l'index où devrait se trouver la chaîne «ab». Remarquez que la fonction hache le pointeur sur la dernière chaîne lue sur le flot d'entrée (C) avec le symbole du dernier caractère lu (D). Supposons que l'on ait obtenu `pointeur := 5`. L'encodeur va maintenant regarder s'il peut trouver la chaîne à l'indice 5 du tableau. Cette case est marquée comme inoccupée. Par conséquent, on sait que le dictionnaire ne contient pas «ab». L'encodeur va donc rajouter cette chaîne :

parent	0	0	/	/	1	···
index	1	2	-	-	5	···
symbole	a	b			b	···

Le contenu de la case 5 est obtenu grâce aux instructions suivantes :

```
dict[pointeur].parent := C ;
dict[pointeur].index := pointeur ;
dict[pointeur].symbole := D ;
```

Dans LZW, lorsqu'une chaîne est insérée dans le dictionnaire, le pointeur C est réinitialisé à la valeur du dernier symbole de cette chaîne. Cela revient à effectuer l'opération $C := D$. Par conséquent, maintenant, C est égal à 2. On peut lire le prochain symbole sur le flot d'entrée. C'est un «a». Donc D prend la valeur 1, c'est-à-dire la valeur de l'index de «a». L'encodeur doit maintenant rechercher la chaîne «ba», dont l'index est déterminé par l'appel à la fonction de hachage : `pointeur := hash(C,D) = hash(2,1)`. Supposons que le résultat soit égal à 8. Cette case étant inoccupée, on sait que la chaîne «ba» n'appartient pas au dictionnaire. On va donc la remplir :

```
dict[pointeur].parent := C = 2 ;
dict[pointeur].index := pointeur = 8 ;
dict[pointeur].symbole := D = 1 ;
```

On obtient alors le dictionnaire suivant :

parent	0	0	/	/	1	/	/	2	···
index	1	2	-	-	5	-	-	8	···
symbole	a	b			b			a	···

La chaîne «ba» ayant été sauvée dans le dictionnaire, il faut réinitialiser C à la valeur de «a», c'est-à-dire 1. On a donc $C = 1$ et, comme le nouveau caractère sur le flot d'entrée est un «b», on a $D = 2$. L'encodeur doit donc rechercher la chaîne «ab» dans le dictionnaire, autrement dit, la chaîne dont l'index devrait être `pointeur := hash(1,2)`, dont le résultat est 5 comme nous l'avions vu. Dans cette case, on a `dict[5].index = 5`, par conséquent, la chaîne «ab» appartient déjà au dictionnaire. On peut donc continuer avec $C = 5$.

Lisons maintenant le cinquième symbole du flot d'entrée. C'est un «a». Donc $D = 1$. L'encodeur doit maintenant rechercher la chaîne «aba» dans le dictionnaire, autrement dit, il doit examiner la case d'index : `pointeur := hash(5,1)`. Supposons que `pointeur` soit égal à 8. Dans ce cas, il y a collision car cette case est occupée, mais pas par «aba» (on s'en aperçoit car `dict[8].parent` $\neq$ `5`). On va alors essayer de voir si «aba» ne pourrait pas se trouver plus loin dans le dictionnaire, à l'index 9, puis en 10, etc, jusqu'à ce qu'on le trouve ou que l'on arrive sur une case non occupée. C'est précisément le cas de la case 9. Cela veut dire que «aba» n'appartient pas au dictionnaire. On peut donc la rajouter à l'index 9. D'où le dictionnaire :

parent	0	0	/	/	1	/	/	2	5	···
index	1	2	-	-	5	-	-	8	8	···
symbole	a	b			b			a	a	···

Comme la chaîne vient d'être sauvegardée dans le dictionnaire, on doit réinitialiser C à la valeur du dernier symbole de la chaîne, autrement dit a. Par conséquent, on continue la lecture du flot d'entrée avec $C = 1$. Et ainsi de suite. ♦

Le décodage est beaucoup plus simple. Comme dans la structure `dico`, on stocke le pointeur sur le parent des noeuds, il est facile de remonter d'un noeud vers la racine. Cela permet de reconstituer des phrases dans

l'ordre inverse de leur apparition dans le texte d'origine (le texte non compressé). En stockant les symboles des noeuds dans une pile LIFO, les dépilements reconstituent la phrase d'origine et peuvent donc être envoyés par le décodeur sur le flot de sortie.

3.5 Quelques logiciels de compression classiques

Les algorithmes que nous avons vu jusqu'à maintenant, qu'ils soient statistiques ou à base de dictionnaires, sont largement utilisés dans les logiciels que l'on trouve dans le commerce. Pour finir cette section, et avant de voir comment sont compressées les images (GIF et JPEG), faisons un bref tour d'horizon de quelques logiciels de compression classiques.

3.5.1 compress

Dans le monde UNIX/Linux sévissent quelques logiciels de compression de données dont `compress` (vous savez les extensions .Z). Cet utilitaire emploie un algorithme à base de dictionnaire appelé LZC. En fait, LZC est une variante de LZW avec un dictionnaire dont la taille peut augmenter. Plus précisément, la taille du dictionnaire au début de la compression est de $2^9 = 512$ entrées. Comme dans LZW, les 256 premières entrées correspondent à l'ensemble des chaînes de 1 symbole. Tant que le dictionnaire n'est pas rempli, on lui rajoute des chaînes exactement comme on le fait dans LZW. De plus, des pointeurs sur 9 bits sont émis sur le flot de sortie. Si le dictionnaire se remplit totalement, alors sa taille est doublée, c'est-à-dire qu'on passe à un dictionnaire ayant 1024 entrées. Tant que ces 1024 entrées ne sont pas remplies, on émet sur le flot de sortie des pointeurs de 10 bits. Lorsque ce nouveau dictionnaire est plein, on redouble à nouveau la taille du dictionnaire. Et ainsi de suite jusqu'à ce que l'on atteigne une taille de $2^{16} = 65536$ entrées. Là, on fige le dictionnaire. Celui-ci devient donc statique pour tout le reste de la compression.

3.5.2 Zip et Gzip

Ces deux programmes implémentent l'algorithme appelé «*deflation*» qui combine une variante de LZ77 en première passe avec une méthode de Huffman en deuxième passe.

Plus précisément, la première passe utilise une fenêtre coulissante, dont la fenêtre dictionnaire est de 32Ko et la fenêtre à compresser de 258 octets. Quand une chaîne est trouvée dans le dictionnaire, un couple de la forme (distance, longueur) est émis sur le flot de sortie (en fait le flot d'entrée de la deuxième passe). Dans le cas contraire, c'est une séquence d'octets non compressés qui est émise.

Le stockage du dictionnaire se fait grâce à une table de hachage. Plus précisément, toutes les chaînes de longueur 3 du flot d'entrée sont insérées dans la table de hachage. Une fonction de hachage permet de calculer un index sur les trois prochains octets. Si la case correspondante dans la table n'est pas vide, toutes les chaînes de la table de hachage sont comparées à la chaîne courante du flot d'entrée et c'est celle qui est identique et qui est la plus longue qui est sélectionnée (c'est vraiment du LZ77). La table de hachage est parcourue en commençant par les chaînes les plus récentes de manière à obtenir les chaînes à la plus petite distance possible. Cela permet en effet de compresser encore mieux quand on utilise Huffman. Les chaînes qui sont trop vieilles sont effacées progressivement de la table de hachage (de manière à simuler la fenêtre coulissante). De plus, afin d'éviter des temps d'exécution trop prohibitifs, les chaînes «très longues» de la table de hachage sont arbitrairement tronquées à une certaine longueur, longueur qui est déterminée grâce aux options passées au démarrage de zip ou de gzip (par exemple zip -1 à zip -9). Conséquence : «*deflation*» ne peut pas toujours trouver la chaîne réellement la plus longue. En principe, cela suffit toutefois pour obtenir un taux de compression relativement raisonnable.

À l'issue de la première passe, Zip et Gzip effectuent une nouvelle compression à base d'arbres de Huffman. Plus précisément, les octets non compressés et les longueurs des chaînes trouvées dans le dictionnaire (second élément des couples issus de la variante de LZ77) sont compressés grâce à un arbre de Huffman, tandis que les distances (premier élément des couples LZ77) sont compressées avec un autre arbre de Huffman. Le flot d'entrée de la deuxième passe est séparé en blocs par l'encodeur. Leur taille est arbitraire sauf pour certains blocs non compressibles qui sont limités à 64Ko. La fin d'un bloc est déterminée lorsque l'algorithme «*deflation*» pense qu'il serait intéressant de créer un nouveau bloc avec un nouvel arbre de Huffman. Les arbres de Huffman sont indépendants d'un bloc à l'autre et sont stockés sous forme compacte en début de chaque bloc.

3.5.3 ARC et PKarc

ARC et PKarc utilisent tous les deux les mêmes méthodes de compression. Sans rentrer dans le détail, ARC permet plusieurs types de compression :

1. pas de compression (c'est une méthode obsolète). Ce n'est pas abhérrant de trouver cela car ARC n'a pas seulement pour mission de compresser, il a aussi une mission d'archivage.
2. *Stored* : le fichier est simplement copié sur le flot de sortie sans réaliser de compression.
3. *Packed* : les codes ASCII sont émis sur le flot de sortie sur sept bits au lieu de huit.
4. *Squeezed* : D'abord on effectue du packing (donc mise sur 7 bits au lieu de 8) puis on effectue une méthode de Huffman.
5. *Crunched* : pas de packing mais utilisation de LZW avec un dictionnaire statique et émission sur 12 bits (obsolète).
6. *Crunched* : packing puis LZW statique sur 12 bits(obsolète).
7. *Crunched* : packing puis LZW avec utilisation d'un algorithme de hachage plus rapide que précédemment (obsolète).
8. *Crunched* : packing puis utilisation d'une variante dynamique de LZW. Les codes LZW ont au début une taille de 9 bits et au fur et à mesure que le dictionnaire se remplit, leur taille augmente jusqu'à un maximum de 12 bits. Des remises à zéro du dictionnaire peuvent être effectuées.
9. *Squashed* : pas de packing mais compression avec un LZW dynamique. La taille des codes LZW varie de 9 bits à 13 bits. Éventuellement, des remises à zéro du dictionnaire peuvent être effectuées.

3.5.4 les utilitaires Pk (PKZip, PKLite, etc)

La famille des PK (PKZip, PKLite, PKUnzip) utilise elle aussi différents modes de compression :

1. *Shrinking* : Cette méthode utilise une version dynamique de LZW, avec mise à jour partielle du dictionnaire. Les codes LZW ont une taille allant de 9 à 13 bits. Cette taille est déterminée par le compresseur et varie de temps en temps. Quand l'encodeur décide une augmentation de la taille, il émet la séquence 256,1. Ce signal avertit le décodeur qu'il doit faire de même. Quand le dictionnaire est rempli, celui-ci n'est pas effacé entièrement. Plutôt, seules les feuilles du dictionnaire le sont. L'encodeur émet alors la séquence 256,2 afin que le décodeur puisse procéder de même. La taille du code LZW n'a pas alors à être augmentée.
2. *Reducing* : Cette méthode fonctionne en deux étapes : tout d'abord une passe RLE est effectuée. Ensuite, c'est une méthode statistique qui est utilisée pour finir la compression.
3. *Imploding* : Là encore, on a une combinaison de deux méthodes : la première étape consiste à éliminer les séquences de caractères qui se répètent en utilisant une fenêtre coulissante (un dictionnaire de 4Ko ou 8Ko). À l'issue de cette étape, on obtient soit des octets non compressés, soit des couples (distance, longueur) lorsque l'on a trouvé des chaînes dans le dictionnaire. La seconde étape compresse les octets non compressés, les distances et les longueurs en utilisant un algorithme de Shannon-Fano (une variante de Huffman) sur chacun d'eux.

3.5.5 LHA et LHArc

ICE, LHArc et LHA sont des logiciels écrits par Haruyasu Yoshizaki. Ils utilisent tous une méthode de Huffman adaptative conjuguée avec une variante de LZSS.

3.6 Algorithmes en langage C

3.6.1 LZ77

```
/* ================================================================================================================ */
/* === CG - prog-sec3-LZ77.c - version du 19/9/2001                                                             === */
/* === realise l'algorithme LZ77 tel que decrit dans la section 3.                                              === */
/* === la fenetre coulissante est geree grace a un tableau circulaire.                                          === */
/* ================================================================================================================ */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define SEARCH_BUFFER_SIZE  16384  /* la fenetre dictionnaire contient au plus 65536 caracteres */
#define LOOK_BUFFER_SIZE      256  /* nombre de caracteres max de la fenetre a compresser */
#define BUFFER_SIZE  (SEARCH_BUFFER_SIZE + LOOK_BUFFER_SIZE)

/* petite macro pour faire des operations modulo BUFFER_SIZE */
#define op(x) (x >= BUFFER_SIZE ? (x) - BUFFER_SIZE : (x < 0 ? (x) + BUFFER_SIZE : (x)))


/* ================================================================================================================ */
/* === structure permettant de stocker un tableau circulaire.                                                   === */
/* ================================================================================================================ */
typedef struct {
  unsigned char *array;  /* le tableau */
           int   search_begin;  /* l'index du premier caractere de la fenetre dictionnaire */
           int   look_begin;    /* l'index du premier caractere de la fenetre a compresser. Normalement, ce */
                                /* caractere se trouve juste a droite du dernier caractere de la fenetre dico. */
                                /* lorsque look_begin == search_begin, le dictionnaire est vide. */
           int   look_end;      /* l'index du premier caractere apres la fenetre a compresser. */
           int   look_size;     /* taille de la fenetre a compresser */
} window;



/* ================================================================================================================ */
/* === prototypes des fonctions contenues dans ce fichier.                                                      === */
/* ================================================================================================================ */
char   *useful_index_bytes(int nb);
window *allocate_window(int size);
void    free_window(window *win);
void    look_for_string(window *win, int *index, int *string_size);
void    advance_window(window *win, int nb_char);
void    read_to_window(window *win, FILE *filein, int nb_char);
void    compresse(char *filenamein, char *filenameout);


/* ================================================================================================================ */
/* === la fonction suivante retourne un tableau de booleens indiquant quels octets de l'int sont                === */
/* === necessaires pour stocker l'entier et quels octets ne servent a rien. On a besoin de cette fonction       === */
/* === pour que le programme fonctionne sur n'importe quelle architecture (small endian, big endian, etc).      === */
/* ================================================================================================================ */
char *useful_index_bytes(int nb)
{
  int   i, nb_decal;
  char *tab, *octet;

  /* on commence par transformer nb en le plus petit nombre nb' de la forme 2^x - 1 tel que nb' >= nb */
  for (i=0, nb--, nb_decal=0; nb != 0 && i<sizeof(int) * 8; i++, nb >>= 1, nb_decal++);
  for (i=0, nb=0; i<nb_decal; i++, nb = (nb << 1) + 1);

  /* maintenant on alloue le tableau de booleens (en fait des char a 1=true ou 0=false */
  tab=malloc(sizeof(int));
  if (tab == NULL)
    {
      fprintf(stderr, "useful_index_bytes : allocation memoire impossible\n");
      return NULL;
    }

  /* on remplit tab : il suffit de mettre des 1 sur les octets != 0 et 0 sur les autres */
  for (i=0, octet=(char*)&nb; i<sizeof(int); i++, octet++)
    if (*octet) tab[i] = 1;
    else tab[i] = 0;

  return tab;
}
```

```
/* ================================================================================================================ */
/* === cette fonction alloue une structure de fenetre circulaire (fenetre dictionnaire + fenetre a         === */
/* === compresser) et la renvoie.                                                                           === */
/* ================================================================================================================ */
window *allocate_window(int size)
{
  window *win;

  /* on teste si la taille de la fenetre est correcte */
  if (size <= 0)
    {
      fprintf(stderr, "allocate_window :impossible d'allouer une fenetre ");
      fprintf(stderr, "de taille negative ou nulle\n");
      return NULL;
    }

  /* on alloue la fenetre et le tableau */
  win = (window *) malloc(sizeof(window));
  if (win == NULL)
    {
      fprintf(stderr, "allocate_window : allocation memoire impossible\n");
      return NULL;
    }
  win->array = (unsigned char *) malloc(size);
  if (win->array == NULL)
    {
      free(win);
      fprintf(stderr, "allocate_window : allocation memoire impossible (2)\n");
      return NULL;
    }

  /* remplissage des champs de la structure et retour de la fonction */
  win->search_begin = 0;
  win->look_begin   = 0;
  win->look_end     = 0;
  win->look_size    = 0;

  return win;
}


/* ================================================================================================================ */
/* === cette fonction desalloue une fenetre circulaire.                                                     === */
/* ================================================================================================================ */
void free_window(window *win)
{
  if ((win != NULL) && (win->array != NULL))
    free (win->array);
  if (win != NULL)
    free (win);
}


/* ================================================================================================================ */
/* === cette fonction cherche dans la fenetre dictionnaire la plus longue chaine prefixe de la fenetre     === */
/* === a compresser. Si aucune chaine n'est trouvee, elle affecte 0 a index et 0 a string_size, sinon,      === */
/* === elle affecte a index l'index du 1er caractere de la chaine prefixe et a string_size la taille de    === */
/* === la chaine prefixe, comme indique dans le cours.                                                      === */
/* ================================================================================================================ */
void look_for_string(window *win, int *index, int *string_size)
{
  int search_index, look_index, ind, ind1, ind2;
  int unmatch, nb;

  /* on balaye la fenetre dictionnaire */
  for(search_index = op(win->look_begin-1), look_index = win->look_begin, *string_size = 1, *index = -1, ind=0;
      search_index != op(win->search_begin-1); search_index = op(search_index-1), ind++)
    {
      /* on regarde si la chaine partant de search_index est une chaine prefixe de la fenetre a compresser */
      for (unmatch=0, nb=0, ind1=search_index, ind2=look_index;
           !unmatch && ind2!=win->look_end; ind1=op(ind1+1), ind2=op(ind2+1), nb++)
        if (win->array[ind1] != win->array[ind2]) unmatch = 1;

      /* on a trouve une plus grande chaine prefixe => on sauvegarde sa place et sa longueur */
      if (nb > *string_size)
        {
          *index = ind;
          *string_size  = nb - 1;
        }
```

```
    }

  if (*index == -1)
    {
      *index = 0;
      *string_size = 0;
    }
}


/* ================================================================================================================ */
/* === la fonction suivante deplace la fenetre dictionnaire de nb_char. ATTENTION : elle ne met pas a jour === */
/* === de maniere correspondante la fenetre a compresser.                                                   === */
/* ================================================================================================================ */
void advance_window(window *win, int nb_char)
{
  int search_size, nb_add;

  /* on calcule la taille de la fenetre dictionnaire actuelle. */
  if (win->look_begin < win->search_begin)
    search_size = BUFFER_SIZE + win->look_begin - win->search_begin;
  else
    search_size = win->look_begin - win->search_begin;

  /* si la fenetre n'a pas encore atteint la taille maximale, on l'agrandit */
  if (search_size < SEARCH_BUFFER_SIZE)
    {
      nb_add = (search_size + nb_char) <= SEARCH_BUFFER_SIZE ? nb_char : SEARCH_BUFFER_SIZE - search_size;
      win->look_begin = op(win->look_begin + nb_add);
      win->look_size -= nb_add;
      nb_char -= nb_add;
      search_size += nb_add;
    }

  /* la fenetre a la taille maximale => on effectue un deplacement */
  if (search_size == SEARCH_BUFFER_SIZE)
    {
      win->search_begin = op(win->search_begin + nb_char);
      win->look_begin   = op(win->look_begin + nb_char);
      win->look_size   -= nb_char;
    }
}


/* ================================================================================================================ */
/* === la fonction suivante lit nb_char (au plus) dans le fichier d'entree et les place dans la fenetre    === */
/* === a compresser. Elle met a jour les indicateurs de fin de fenetre et de taille.                        === */
/* ================================================================================================================ */
void read_to_window(window *win, FILE *filein, int nb_char)
{
  unsigned char str[LOOK_BUFFER_SIZE];
           int  nb_read, nb_max;

  nb_read = fread(str, 1, nb_char, filein);
  if (nb_read == 0) return;

  nb_max = BUFFER_SIZE - win->look_end;
  nb_max = nb_max < nb_read ? nb_max : nb_read;

  memmove(win->array + win->look_end, str, nb_max);
  if (nb_max < nb_read) memmove(win->array, str+nb_max, nb_read - nb_max);
  win->look_size += nb_read;
  win->look_end = op(win->look_end + nb_read);
}


/* ================================================================================================================ */
/* === comme son nom l'indique, cette fonction compresse le fichier filenamein dans le fichier filenameout === */
/* ================================================================================================================ */
void compresse(char *filenamein, char *filenameout)
{
           FILE   *in, *out;
           window *win;
           char   *index_bytes, *string_size_bytes, *ind, *size;
           int     index, string_size;
  unsigned char    print_string[2 * sizeof(int) + 1];
           int     i, j;

  /* ouverture des deux fichiers et allocation de la fenetre circulaire */
```

```
  in = fopen(filenamein, "rb");
  if (in == NULL)
    {
      fprintf(stderr, "compresse : impossible d'ouvrir le fichier %s\n", filenamein);
      exit(EXIT_FAILURE);
    }
  out = fopen(filenameout, "wb");
  if (out == NULL)
    {
      fclose(in);
      fprintf(stderr, "compresse : impossible (2) d'ouvrir le fichier %s\n", filenameout);
      exit(EXIT_FAILURE);
    }
  win = allocate_window(BUFFER_SIZE);
  if (win == NULL)
    {
      fclose(in); fclose(out);
      fprintf(stderr, "compresse : impossible d'allouer la fenetre coulissante\n");
      exit(EXIT_FAILURE);
    }

  /* on calcule quels octets vont etre ecrits utilises pour sauvegarder les 2 premiers elements des triplets */
  index_bytes = useful_index_bytes(SEARCH_BUFFER_SIZE);
  string_size_bytes = useful_index_bytes(LOOK_BUFFER_SIZE);

  /* 1ere lecture du fichier d'entree => remplissage de la fenetre a compresser */
  read_to_window(win, in, LOOK_BUFFER_SIZE);

  /* ecriture de l'en-tete du fichier */
  fprintf(out, "LZ77 - V1.0");

  /* ecriture du fichier compresse */
  while (win->look_size > 0)
    {
      /* recherche dans la fenetre dictionnaire et ecriture des triplets */
      look_for_string(win, &index, &string_size);
      for (i=0, j=0, ind = (char*)&index; i<sizeof(int); i++)
        if (index_bytes[i]) print_string[j++] = ind[i];
      for (i=0, size=(char *)&string_size; i<sizeof(int); i++)
        if (string_size_bytes[i]) print_string[j++] = size[i];
      print_string[j++] = win->array[op(win->look_begin + string_size)];
      fwrite(print_string, j, 1, out);

      /* deplacement de la fenetre circulaire */
      advance_window(win, string_size+1);
      read_to_window(win, in, string_size+1);
    }

  /* fermeture des fichiers : la compression est terminee */
  free_window(win);
  fclose(in);
  fclose(out);
}


/* ================================================================================================================ */
/* === la fonction ci-dessous decompresse le fichier filenamein dans le fichier filenameout.                  === */
/* ================================================================================================================ */
void decompresse(char *filenamein, char *filenameout)
{
           FILE   *in, *out;
           window *win;
           char   *index_bytes, *string_size_bytes;
           int     index=0, string_size=0;
           int     i, j;
           int     nb_read, index_bytes_nb, string_size_bytes_nb, nb_octets;
  unsigned char en_tete[20], car, *ptr, string[sizeof(int)], str[LOOK_BUFFER_SIZE], *str_out;
  unsigned char *tab_in, *tab_out;

  /* ouverture des fichiers et allocation de la fenetre a compresser */
  in = fopen(filenamein, "rb");
  if (in == NULL)
    {
      fprintf(stderr, "decompresse : impossible d'ouvrir le fichier %s\n", filenamein);
      exit(EXIT_FAILURE);
    }
  out = fopen(filenameout, "wb");
  if (out == NULL)
    {
```

```
    fclose(in);
    fprintf(stderr, "decompresse : impossible (2) d'ouvrir le fichier %s\n", filenameout);
    exit(EXIT_FAILURE);
  }
win = allocate_window(BUFFER_SIZE);
if (win == NULL)
  {
    fclose(in); fclose(out);
    fprintf(stderr, "decompresse : impossible d'allouer la fenetre coulissante\n");
    exit(EXIT_FAILURE);
  }

/* on calcule quels octets des int ont ete sauvegardes dans le fichier d'entree et combien d'octets */
/* sont utilises pour chaque int (les 2 premiers elements des triplets). */
index_bytes = useful_index_bytes(SEARCH_BUFFER_SIZE);
string_size_bytes = useful_index_bytes(LOOK_BUFFER_SIZE);
for (i=0,index_bytes_nb=0, string_size_bytes_nb=0; i<sizeof(int); i++)
  {
    if (index_bytes[i]) index_bytes_nb++;
    if (string_size_bytes[i]) string_size_bytes_nb++;
  }

/* lecture de l'en-tete du fichier */
nb_read = fread(en_tete, 11, 1, in);
if ((!nb_read) || strncmp(en_tete,"LZ77 - V1.0", 11))
  {
    free_window(win); fclose(in); fclose(out);
    fprintf(stderr, "decompresse : le fichier %s n'a pas ete compresse par LZ77\n", filenamein);
    exit(EXIT_FAILURE);
  }

/* ecriture du fichier decompresse */
while (1)
  {
    /* lecture d'un triplet */
    nb_read = fread(string, 1, index_bytes_nb, in);
    if (feof(in))
      {
        free_window(win); fclose(in); fclose(out);
        return;
      }
    if (nb_read != index_bytes_nb)
      {
        free_window(win); fclose(in); fclose(out);
        fprintf(stderr, "decompresse (2) : le fichier %s n'a pas ete compresse par LZ77\n", filenamein);
        exit(EXIT_FAILURE);
      }
    for (i=0, j=0, ptr=(char*)&index; i<sizeof(int); i++)
      if (index_bytes[i]) ptr[i] = string[j++];
    nb_read = fread(string, 1, string_size_bytes_nb, in);
    if (nb_read != string_size_bytes_nb)
      {
        free_window(win); fclose(in); fclose(out);
        fprintf(stderr, "decompresse (3) : le fichier %s n'a pas ete compresse par LZ77\n", filenamein);
        exit(EXIT_FAILURE);
      }
    for (i=0, j=0, ptr=(char*)&string_size; i<sizeof(int); i++)
      if (string_size_bytes[i]) ptr[i] = string[j++];
    nb_read = fread(&car, 1, 1, in);
    if (!nb_read)
      {
        free_window(win); fclose(in); fclose(out);
        fprintf(stderr, "decompresse (4) : le fichier %s n'a pas ete compresse par LZ77\n", filenamein);
        exit(EXIT_FAILURE);
      }

    /* ecriture du texte dans le fichier de sortie ainsi que dans la fenetre dictionnaire */
    for (tab_in=win->array+win->look_begin, tab_out=win->array+op(win->look_begin-index-1),
           str_out=str, nb_octets=string_size;
         nb_octets; nb_octets--)
      {
        *tab_in = *tab_out;
        *str_out = *tab_out;
        str_out++;
        if (++tab_in  == win->array+BUFFER_SIZE) tab_in  = win->array;
        if (++tab_out == win->array+BUFFER_SIZE) tab_out = win->array;
      }
    *tab_in  = car;
    *str_out = car;
```

```
      fwrite(str, string_size+1, 1, out);

      /* mise a jour des index de la fenetre coulissante */
      advance_window(win, string_size+1);
      win->look_end=op(win->look_end+string_size+1);
      win->look_size+=string_size+1;
    }

  free_window(win);
  fclose(in);
  fclose(out);
}


/* ================================================================================================================ */
/* === le programme principal attend trois arguments. Le premier indique si l'on doit compresser ou        === */
/* === decompresser un fichier. le second est le nom du fichier sur lequel va porter l'action, et le       === */
/* === troisieme est le nom du fichier dans lequel sera stocke le resultat de l'action                     === */
/* ================================================================================================================ */
int main(int argc, char **argv)
{
  /* test des arguments */
  if ((argc != 4) ||
      (strcmp(argv[1], "compresse") && strcmp(argv[1], "decompresse")))
    {
      fprintf(stderr, "usage : prog-sec3-LZ77 action fic_entree fic_sortie\n");
      fprintf(stderr, "        action doit etre egal soit a la chaine");
      fprintf(stderr, " compresse soit a decompresse\n");
      exit (EXIT_FAILURE);
    }

  /* on effectue l'action demandee */
  if (!strcmp(argv[1], "compresse"))
    compresse(argv[2], argv[3]);
  else
    decompresse(argv[2], argv[3]);

  return EXIT_SUCCESS;
}
```

3.6.2 LZ78

```
/* ================================================================================================================ */
/* === CG - prog-sec3-LZ78.c - version du 19/9/2001                                                         === */
/* === realise l'algorithme LZ78 tel que decrit dans le cours. Lorsque le tableau (arbre) est plein, il    === */
/* === est reinitialise a 0.                                                                                === */
/* ================================================================================================================ */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define DICO_SIZE 16384

/* ================================================================================================================ */
/* === les structures contenant le dictionnaire. En fait, c'est un arbre stocke dans un tableau. La        === */
/* === structure d'arbre permet d'obtenir un algorithme tres rapide, et le tableau evite les allocations   === */
/* === et desallocations intempestives.                                                                     === */
/* ================================================================================================================ */
typedef struct _arbre arbre;

struct _arbre {
  arbre *parent;       /* le parent du noeud courant */
  arbre *next;         /* le fils suivant du parent : tous les fils sont chaines entre eux */
  arbre *first_child;  /* le premier fils du noeud courant, les autres seront accessibles grace au champ */
                       /* next du fils. */
  unsigned char car;   /* le caractere correspondant au 2eme element des couples LZ78 */
};

typedef struct {
  arbre *dico;         /* un dico est en fait un arbre (un tableau de noeuds) */
  int    index;        /* l'index du dernier noeud utilise dans le tableau, permet de savoir ou l'on */
} dictionnaire;        /* devra placer le prochain noeud cree dans l'arbre */
```

```
/* ================================================================================================================ */
/* === la fonction suivante retourne un tableau de booleens indiquant quels octets de l'int sont           === */
/* === necessaires pour stocker l'entier et quels octets ne servent a rien. On a besoin de cette fonction  === */
/* === pour que le programme fonctionne sur n'importe quelle architecture (small endian, big endian, etc). === */
/* ================================================================================================================ */
char *useful_index_bytes(int nb)
{
  int   i, nb_decal;
  char *tab, *octet;

  /* on commence par transformer nb en le plus petit nombre nb' de la forme 2^x - 1 tel que nb' >= nb */
  for (i=0, nb--, nb_decal=0; nb != 0 && i<sizeof(int) * 8; i++, nb >>= 1, nb_decal++);
  for (i=0, nb=0; i<nb_decal; i++, nb = (nb << 1) + 1);

  /* maintenant on alloue le tableau de booleens (en fait des char a 1=true ou 0=false */
  tab=malloc(sizeof(int));
  if (tab == NULL)
    {
      fprintf(stderr, "useful_index_bytes : allocation memoire impossible\n");
      return NULL;
    }

  /* on remplit tab : il suffit de mettre des 1 sur les octets != 0 et 0 sur les autres */
  for (i=0, octet=(char*)&nb; i<sizeof(int); i++, octet++)
    if (*octet) tab[i] = 1;
    else tab[i] = 0;

  return tab;
}


/* ================================================================================================================ */
/* === la fonction suivante alloue et initialise un nouveau dictionnaire.                                     === */
/* ================================================================================================================ */
dictionnaire *create_dico()
{
  dictionnaire *dico;

  dico = (dictionnaire *)malloc(sizeof(dictionnaire));
  if (dico == NULL)
    {
      fprintf(stderr, "create_dico : allocation memoire impossible\n");
      return NULL;
    }

  dico->dico = (arbre *)malloc(DICO_SIZE * sizeof(arbre));
  if (dico->dico == NULL)
    {
      fprintf(stderr, "create_dico : allocation memoire impossible (2)\n");
      return NULL;
    }

  dico->index = 1;

  return dico;
}


/* ================================================================================================================ */
/* === desallocation du dictionnaire pour faire plus propre en fin de compression.                            === */
/* ================================================================================================================ */
void free_dico(dictionnaire *dico)
{
  if ((dico == NULL) || (dico->dico == NULL))
    {
      fprintf(stderr, "free_dico : ne peut pas desallouer un dictionnaire vide\n");
      return;
    }

  free(dico->dico);
  free(dico);
}
```

```
/* ================================================================================================================ */
/* === la fonction suivante part d'un noeud de l'arbre (qui est donc l'equivalent d'une chaine de          === */
/* === caracteres, selon LZ78) et elle recherche si la concatenation de cette chaine et d'un caractere c   === */
/* === existe dans le dictionnaire. Le cas echeant, elle renvoie le noeud correspondant de l'arbre, sinon  === */
/* === elle renvoie le pointeur NULL.                                                                      === */
/* ================================================================================================================ */
arbre *look_in_dico(dictionnaire *dico, arbre *start, char c)
{
  /* en phase de debug, on s'assure qu'on passe bien un dictionnaire en parametre */
  assert((dico != NULL) && (dico->dico != NULL));

  /* on regarde de quel noeud on doit partir : si start == NULL, c'est qu'on part du debut du dico, sinon, */
  /* c'est qu'on a deja atteint start et on doit visiter ses enfants */
  for (start = (start == NULL ? dico->dico + 1 : start->first_child); start != NULL; start = start->next)
    if (start->car == c) return start;

  return NULL;
}


/* ================================================================================================================ */
/* === la fonction suivante rajoute un nouveau noeud dans l'arbre : le pere de ce noeud est 'start', sauf  === */
/* === si celui-ci est NULL, auquel cas le noeud devient la racine de l'arbre. Si le dictionnaire est      === */
/* === plein, la fonction le vide puis rajoute le noeud, qui devient alors racine du nouveau dictionnaire. === */
/* ================================================================================================================ */
void add_in_dico(dictionnaire *dico, arbre *start, char c)
{
  arbre *noeud;

  /* en phase de debug, on s'assure qu'on passe bien un dictionnaire en parametre */
  assert((dico != NULL) && (dico->dico != NULL));

  /* on regarde si on est arrive a la fin du dictionnaire => dans ce cas, on repart d'un dico vide */
  if (dico->index == DICO_SIZE)
    {
      start = NULL;
      dico->index = 1;
    }

  /* ajout du nouveau noeud dans l'arbre */
  noeud = dico->dico + dico->index;
  noeud->first_child = NULL;
  noeud->car = c;
  noeud->parent = start;
  /* si on part d'un start non vide, il faut le noeud raccorder aux fils du noeud start */
  if (start != NULL)
    {
      noeud->next = start->first_child;
      start->first_child = noeud;
    }
  else
    {
      if (dico->index)
        {
          for (start = dico->dico + 1; start->next != NULL; start = start->next);
          start->next = noeud;
        }
      noeud->next = NULL;
    }

  /* on incremente le compteur de noeuds affectes */
  dico->index++;
}


/* ================================================================================================================ */
/* === compression en utilisant LZ78. La fonction place en en-tete du fichier compresse: "LZ78 - V1.0".    === */
/* ================================================================================================================ */
void compresse(char *filenamein, char *filenameout)
{
  FILE *in, *out;
  dictionnaire *dico;
  arbre *start, *node;
  char car;
  char *index_bytes, *ind, print_string[sizeof(int) + 1];
  int  index;
  int  i, j;

  /* ouverture des deux fichiers et allocation du dictionnaire */
```

```
  in = fopen(filenamein, "rb");
  if (in == NULL)
    {
      fprintf(stderr, "compresse : impossible d'ouvrir le fichier %s\n", filenamein);
      exit(EXIT_FAILURE);
    }
  out = fopen(filenameout, "wb");
  if (out == NULL)
    {
      fclose(in);
      fprintf(stderr, "compresse : impossible (2) d'ouvrir le fichier %s\n", filenameout);
      exit(EXIT_FAILURE);
    }
  dico = create_dico();
  if (dico == NULL)
    {
      fclose(in);
      fclose(out);
      exit(EXIT_FAILURE);
    }

  /* ecriture de l'en-tete du fichier */
  fprintf(out, "LZ78 - V1.0");

  /* compression du fichier */
  start = NULL;
  index_bytes = useful_index_bytes(DICO_SIZE);
  while (fread(&car, 1, 1, in) == 1)
    {
      /* recherche de la lettre dans le dico et ajout si elle n'est pas trouvee */
      node = look_in_dico(dico, start, car);
      if (node == NULL)
        {
          /* ecriture du couple dans le fichier de sortie */
          if (start == NULL)
            index = 0;
          else
            index = start - dico->dico;
          for (i=0, j=0, ind = (char*)&index; i<sizeof(int); i++)
            if (index_bytes[i]) print_string[j++] = ind[i];
          print_string[j++] = car;
          fwrite(print_string, j ,1, out);

          /* ajout dans le dictionnaire */
          add_in_dico(dico, start, car);
          start = NULL;
        }
      else
        start = node;
    }

  /* on risque d'avoir a ecrire old_start + car sur le fichier si node != NULL */
  if (node != NULL)
    {
      if (node->parent == NULL)
        index = 0;
      else
        index = node->parent - dico->dico;
      for (i=0, j=0, ind = (char*)&index; i<sizeof(int); i++)
        if (index_bytes[i]) print_string[j++] = ind[i];
      print_string[j++] = car;
      fwrite(print_string, j ,1, out);
    }

  /* fermeture des fichiers : la compression est terminee */
  free_dico(dico);
  fclose(in);
  fclose(out);
}
```

```
/* ================================================================================================================== */
/* === decompression d'un fichier compresse avec LZ78. Attention a l'en-tete "LZ78 - V1.0".                  === */
/* ================================================================================================================== */
void decompresse(char *filenamein, char *filenameout)
{
  FILE *in, *out;
  dictionnaire *dico;
  arbre *node;
  int index_bytes_nb, nb_read, nb, index=0;
  char *index_bytes, en_tete[11], string[sizeof(int)+1], print_string[DICO_SIZE+1], *ptr, car;
  int  i,j;

  /* ouverture des deux fichiers et allocation du dictionnaire */
  in = fopen(filenamein, "rb");
  if (in == NULL)
    {
      fprintf(stderr, "decompresse : impossible d'ouvrir le fichier %s\n", filenamein);
      exit(EXIT_FAILURE);
    }
  out = fopen(filenameout, "wb");
  if (out == NULL)
    {
      fclose(in);
      fprintf(stderr, "decompresse : impossible (2) d'ouvrir le fichier %s\n", filenameout);
      exit(EXIT_FAILURE);
    }
  dico = create_dico();
  if (dico == NULL)
    {
      fclose(in);
      fclose(out);
      exit(EXIT_FAILURE);
    }

  /* compression du fichier */
  index_bytes = useful_index_bytes(DICO_SIZE);
  for (i=0,index_bytes_nb=0; i<sizeof(int); i++)
    if (index_bytes[i]) index_bytes_nb++;

  /* lecture de l'en-tete du fichier */
  nb_read = fread(en_tete, 11, 1, in);
  if ((!nb_read) || strncmp(en_tete,"LZ78 - V1.0", 11))
    {
      free_dico(dico); fclose(in); fclose(out);
      fprintf(stderr, "decompresse : le fichier %s n'a pas ete compresse par LZ78\n", filenamein);
      exit(EXIT_FAILURE);
    }

  while (fread(&string, index_bytes_nb+1, 1, in) == 1)
    {
      /* recuperation de couple (index,caractere) du dictionnaire */
      for (i=0, j=0, ptr=(char*)&index; i<sizeof(int); i++)
        if (index_bytes[i]) ptr[i] = string[j++];
      car = string[j];

      printf("(%d,%c)\n", index, car);

      /* ajout dans le dico et sauvegarde sur fichier */
      if (index == 0)
        fwrite(&car, 1, 1, out);
      else
        {
          node = dico->dico + index;
          for (i = DICO_SIZE-2, nb=1, print_string[DICO_SIZE-1]=car; node!=NULL; i--, node=node->parent, nb++)
            print_string[i] = node->car;
          fwrite(print_string + i+1, 1, nb, out);
        }
      add_in_dico(dico, index == 0 ? NULL : dico->dico + index, car);
    }

  /* fermeture des fichiers : la compression est terminee */
  free_dico(dico);
  fclose(in);
  fclose(out);
}
```

```
/* ================================================================================================================ */
/* === le programme principal attend trois arguments. Le premier indique si l'on doit compresser ou        === */
/* === decompresser un fichier. le second est le nom du fichier sur lequel va porter l'action, et le       === */
/* === troisieme est le nom du fichier dans lequel sera stocke le resultat de l'action                     === */
/* ================================================================================================================ */
int main(int argc, char **argv)
{
  /* test des arguments */
  if ((argc != 4) ||
      (strcmp(argv[1], "compresse") && strcmp(argv[1], "decompresse")))
    {
      fprintf(stderr, "usage : prog-sec3-LZ78 action fic_entree fic_sortie\n");
      fprintf(stderr, "        action doit etre egal soit a la chaine");
      fprintf(stderr, " compresse soit a decompresse\n");
      exit (EXIT_FAILURE);
    }

  /* on effectue l'action demandee */
  if (!strcmp(argv[1], "compresse"))
    compresse(argv[2], argv[3]);
  else
    decompresse(argv[2], argv[3]);

  return EXIT_SUCCESS;
}
```

3.7 Exercices

Exercice 1 Compressez avec LZ77 le texte suivant :

«abaacacabaaabacdddddaacb»

en utilisant une fenêtre dictionnaire de 8 caractères et une fenêtre à compresser de 4 caractères.
Quel est le taux de compression obtenu ?
Compressez avec Huffman et comparez.

Exercice 2 Compressez avec LZSS le texte suivant :

«aabaacaababbacaac»

en utilisant un arbre ayant au plus 8 feuilles (ce qui nous donne un déplacement sur 3 bits) et une fenêtre à compresser sur 4 caractères. Vous pourrez supposer qu'au début de l'algorithme, les préfixes des chaînes du dictionnaire contiennent autant de blancs que nécessaire.
Compressez avec LZ77 et comparez.

Exercice 3 Décompressez le texte suivant, compressé avec un LZ77 ayant une fenêtre dictionnaire de 8 caractères et une fenêtre à compresser de 4 caractères :

«(0,0,a)(1,1,b)(5,2,a)(6,3,b)(0,0,c)(2,3,a)(7,3,a)(0,0,d)(1,3,a)(6,1,EOF)».

Calculez le taux de compression obtenu en supposant que les déplacements, les longueurs et les lettres sont toutes codées sur 3 bits.
Appliquez sur le texte compressé ci-dessus l'algorithme de Huffman. Est-il normal que le taux de compression augmente ?

Exercice 4 Soit le texte suivant compressé par LZ78 :

«(0,a)(1,b)(0,b)(1,a)(3,a)(4,b)(2,b)(0,c)(1,c)(8,b)(10,b)».

1/ décompressez ce texte.
2/ calculez le taux de compression obtenu par LZ78.

Exercice 5 Compressez à l'aide de LZ78 le texte suivant : «abababbaaabaaaaaababaa». Vous écrirez le dictionnaire ainsi que le flot de sortie.

Si, dans les couples émis, les pointeurs font 12 bits et les symboles 8 bits, quel est le taux de compression réalisé ?

Exercice 6 Décompressez à l'aide de LZ78 le texte suivant :

«(0,a)(1,a)(2,b)(1,b)(4,b)(2,a)(5,a)(6,a)».

Recompressez ce texte avec Huffman. Lequel des deux algorithmes compresse le mieux ce texte ? Pourquoi ?

Exercice 7 Soit l'alphabet {a,b,c,d,EOF}. Compressez avec LZW le texte suivant :

«abbabbbacababaabcEOF».

Quel est le taux de compression ? Comparez avec une compression LZ77 avec une fenêtre dictionnaire de 32KO et une fenêtre à compresser de 256 octets.

Exercice 8 Soit le texte généré par l'algorithme suivant :

```
unsigned char i,j;

for (j='a'; j < 'z'; j++)
  for (i=0; i<128; i++)
    printf("%c", j);
```

Comparez les taux de compression obtenus par un LZ77 avec une fenêtre dictionnaire de 32KO et une fenêtre à compresser de 256 octets, et par un LZ78 avec un dictionnaire ayant au plus 32768 entrées.

Exercice 9 Décompressez le texte suivant, compressé grâce à LZW en utilisant l'alphabet {a,b,c,d,EOF} :

«0 0 1 6 1 9 7 6 8 6 EOF».

4 La compression d'images

Ces 25 dernières années, l'accès aux ordinateurs s'est peu à peu démocratisé. Depuis l'ENIAC (**E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer), le premier ordinateur digital électronique développé par l'université de Pennsylvanie et l'armée américaine pour effectuer des calculs balistiques pendant la seconde guerre mondiale, que de chemin parcouru! De nos jours, les ordinateurs ne sont plus exclusivement consacrés à la réalisation de «gros» calculs, mais ils servent aussi beaucoup pour la bureautique, les jeux, etc. Actuellement, plus personne ne voudrait travailler sur un ordinateur qui ne possède pas d'environnement graphique (comme Windaube ou X11). Or, dans ces environnements, se pose un gros problème : comment stocker efficacement des images (que ce soient des icones, des fonds d'écran, etc). Par efficacement, on entend deux aspects bien différents : i) il faut que les images n'utilisent pas trop de capacité de stockage (il faut bien voir qu'une petite image bitmap de 200×200 sur 65536 couleurs nécessite déjà près de 80KO. Songez alors à ce que consommerait l'image bitmap du cours de probabilité ou du cours de compression de données : mon disque dur ne suffirait pas pour stocker ces cours!). Cet aspect est d'autant plus important avec l'utilisation d'internet car il faut pouvoir transférer de plus en plus rapidement des quantités astronomiques d'images ; ii) si l'on arrive à compresser une image, il faut que la décompression soit rapide.

Il existe deux grandes familles d'algorithmes de compression d'images : les méthodes conservatives et les méthodes non conservatives. Nous verrons deux exemples des premières dans la sous-section 4.1, à savoir GIF et JPEG conservatif, et un exemple des dernières avec JPEG dans la sous-section 4.3. Les méthodes non conservatives sont fondées sur le principe que l'oeil et le cerveau humain ne sont pas capables d'enregistrer tous les détails d'une image. Prenez par exemple le «carré noir» de Kasimir Malevitch, une toile d'environ un mètre sur un mètre qui, comme son nom l'indique fort justement, ne contient qu'un gros carré noir. Rajoutez au centre de la toile un minuscule point gris. Votre oeil ne fera pas la différence entre la «sublime» toile d'origine et l'œuvre immonde que vous venez de créer. Or, la toile d'origine est plus «pauvre» au niveau quantité d'information, donc elle se compresse mieux (entropie plus petite). C'est l'idée sous-jacente des méthodes de compression non conservatives : en éliminant des informations peu ou pas perceptibles par l'oeil, on peut obtenir des taux de compression impressionnants. Comme leur nom l'indique, les méthodes conservatives, elles, ne perdent aucune information et, par là-même, elles sont limitées dans le taux de compression qu'elles peuvent atteindre. Cela dit, grâce à l'utilisation très répandue de GIF, elles ont encore de l'avenir devant elles.

4.1 La compression conservative

Il existe un certain nombre d'algorithmes de compression d'images conservatifs. Ils sont principalement fondés sur les méthodes que nous avons décrites dans les sections précédentes (Huffman, codage arithmétique, RLE, etc) et ont été développés vers la fin des années 80. Depuis, on leur a préféré largement des algorithmes non conservatifs. Toutefois, parmi les formats conservatifs, on peut citer des standards tels que BMP (c'est du bitmap, mais il y a tout de même une compression RLE), PCX, GIF (que nous verrons plus loin).

On peut aussi citer JBIG (le **J**oint **B**i-level **I**mage processing **G**roup). Ce standard est dédié aux images noir et blanc. Il utilise un algorithme de compression progressif, c'est-à-dire que l'image est compressé en plusieurs couches ayant des résolutions de plus en plus grandes. C'est exactement le type de technique utilisée lorsque vous visualisez une image sur votre browser et que vous la voyez apparaître sous forme de gros carrés, puis ceux-ci sont transformés an carrés plus fin, et ainsi de suite jusqu'à l'obtention de l'image finale. Pour chaque couche, JBIG utilise un codage arithmétique. Certains formats d'image disposent de plusieurs algorithmes de compressions. C'est le cas de TIFF (le **T**ag(ged) **I**mage **F**ile **F**ormat) qui peut utiliser PackBits, une variation de RLE, LZW ou bien encore une variation de Huffman.

Pour finir ce petit panorama, remarquons que les algorithmes mentionnés jusqu'à maintenant utilisent de petites variations des techniques développées dans les sections précédentes. CALIC, l'acronyme de **C**ontext-based **A**daptive **L**ossless **I**mage **C**ompression, utilise une idée vraiment différente, qui peut se voir comme une généralisation de RLE : l'idée consiste à utiliser le contexte dans lequel se trouve un pixel pour réaliser une meilleure compression. Cette idée sera développée en détail dans la sous-section 4.1.2, mais disons simplement que, dans une image, un pixel donné est rarement très différent de ses voisins. Donc, si l'on connaît ses voisins, on peut en donner une bonne estimation. CALIC utilise alors un codage arithmétique pour encoder les différences entre les pixels réels de l'image et les estimations réalisées. Je n'en dirai pas plus sur CALIC, simplement, les idées force de CALIC sont assez similaires à celles de LOCO-I que nous verrons dans la sous-section juste

apres GIF. Dans le même ordre d'idée, FELICS, le **F**ast **E**fficient **L**ossless **I**mage **C**ompression **S**ystem, est un algorithme permettant de compresser des images en teintes de gris, qui utilise un contexte de 2 pixels. FELICS compare les intensités des pixels du contexte à celle du pixel courant et en déduit un encodage grâce à un code de taille variable ressemblant à du Huffman.

Le nombre d'algorithmes de compression d'images étant très élevé, et ce cours étant, comme son nom l'indique assez court, nous n'aurons pas le temps de traiter en détails tous ces algorithmes. J'ai donc choisi de ne traiter de manière approfondie que GIF et JPEG, qui me semblent assez représentatifs de ce qui se fait traditionnellement en compression d'image.

4.1.1 La compression sous GIF

GIF est l'acronyme de **G**raphics **I**nterchange **F**ormat. C'est un protocole développé par CompuServe pour permettre l'échange et la visualisation des images, et ce indépendamment de l'architecture sur laquelle on travaille. Le format GIF est défini en termes de blocs et de sous-blocs, qui contiennent les données et paramètres utilisés pour la reproduction de l'image ou des images. En effet, GIF a été développé de manière à pouvoir stocker plusieurs images dans un seul fichier, permettant de créer ainsi des animations. Le format général d'un fichier GIF est décrit dans la figure 10. Le format suppose que les données transmises au décodeur sont sans erreur.

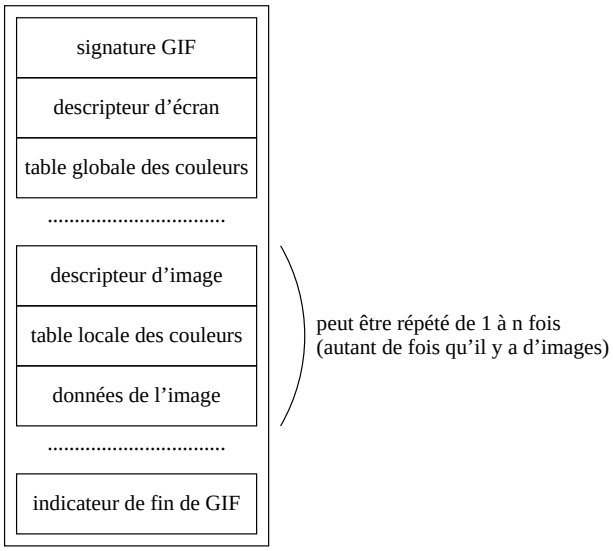


FIG. 10 – La forme générale d'un fichier GIF.

Par conséquent, dans le format GIF, il n'est prévu nulle part de stocker des données redondantes permettant la correction d'erreurs éventuelles (comme les codes correcteurs par exemple). On ne s'intéressera dans le cadre de ce cours qu'à la partie «données de l'image» car c'est elle qui contient la partie «compression de données». Pour une explication détaillée des autres blocs, on peut par exemple se reporter aux pages WEB suivantes :

- `http ://members.aol.com/royalef/gifabout.htm`
- `http ://www.w3.org/Graphics/GIF/spec-gif89a.txt`

Le bloc de données d'une image est constitué d'un ensemble de sous-blocs de tailles inférieures ou égales à 255 octets chacun. Dans les sous-blocs, les pixels sont rangés consécutivement de la gauche vers la droite, puis du haut vers le bas. Ils sont stockés sous forme d'index vers la table active des couleurs (en fait, dans un fichier GIF, il y a une table des couleurs globale qui contient les couleurs partagées par la plupart des images du fichier, et des tables locales qui contiennent les couleurs spécifiques à une image donnée. Dans le descripteur d'image, GIF indique quelle est la table active, c'est-à-dire celle qui est utilisée pour coder les sous-blocs). Enfin, les index sont encodés grâce à un algorithme LZW avec des codes de longueur variables.

L'algorithme GIF-LZW est relativement similaire à celui décrit dans la sous-section 3.4. En voici une explication détaillée. Tout d'abord rappelons que, dans LZW, l'encodeur construit une table des chaînes de caractères (pixels) déjà rencontrées et transmet l'index de chaque chaîne sur le flot de sortie. Pour cela, si le tableau est de taille finie n, il suffirait d'envoyer sur le flot de sortie une représentation binaire de chaque index sur

$\lceil \log_2 n \rceil$ bits. Mais, si le dictionnaire a une taille décente (de l'ordre de 2^{10} à 2^{14} entrées), il va y avoir un bon nombre de bits à 0 au début de l'algorithme, et donc le taux de compression ne va pas être génial. Aussi, GIF applique-t-il une stratégie à base de codes de longueurs variables. Au début de la compression d'une image, GIF garde en mémoire la profondeur N de l'image (le nombre de bits/pixel) et sauvegarde ce nombre en début du bloc de l'image (ce que les anglais appellent le *LZW Minimum Code Size*). Un dictionnaire initial est alors créé, qui contient 2^N entrées et, à l'intérieur, les chaînes sont indexées de 0 à $2^N - 1$. Par exemple, si l'on a une image en noir et blanc (donc d'une profondeur de 1 bit), la table contiendra à l'entrée numéro 0 la chaîne correspondant au pixel blanc, et à l'index 1 celle du pixel noir. En plus de ces index, GIF rajoute au dictionnaire de taille 2^N deux codes spéciaux d'index respectifs 2^N et $2^N + 1$ dont nous verrons plus loin la signification. Comme, dans GIF, les tailles des dictionnaires sont des puissances de 2, lorsqu'on alloue le dictionnaire initial, on réserve en mémoire $2^{\lceil \log_2(2^N+2) \rceil} = 2^{N+1}$ entrées. Au fur et à mesure que l'encodeur rencontre des chaînes de caractères, il remplit le dictionnaire. La première chaîne ainsi rencontrée qui n'appartient pas au dictionnaire sera sauvegardée à l'index $2^N + 2$ et ainsi de suite. Lorsque le dictionnaire est rempli, c'est-à-dire que la dernière chaîne a été entrée à l'index $2^{N+1} - 1$, l'encodeur double la taille du dictionnaire, qui passe donc à 2^{N+2} entrées. Les index qui sont écrits sur le flot de sortie (c'est tout de même le but de LZW) ont le même nombre de bits que le $\log_2$ de la taille du dictionnaire. Ainsi, au début de la compression, puisque ce dernier a une taille de 2^{N+1} entrées, les index sont codés en représentation binaire normale sur $N + 1$ bits. Lorsque la taille du dictionnaire double, la taille de l'index augmente d'un bit.

GIF augmente la taille du dictionnaire quand c'est nécessaire, mais il s'interdit tout de même de l'augmenter indéfiniment afin de ne pas saturer la mémoire. C'est pourquoi, lorsque le dictionnaire possède $2^{12} = 4096$ entrées, GIF décide de ne plus doubler sa taille. Cela dit, il ne peut pas non plus garder très longtemps un dictionnaire statique car, alors, la compression devient de moins en moins bonne (c'est l'idée que les chaînes qui se répètent souvent ne sont pas très éloignées les unes des autres). Donc, lorsque le dictionnaire est rempli au maximum (4096 entrées), GIF choisit simplement de continuer l'encodage en repartant d'un nouveau dictionnaire de la taille initiale 2^{N+1}. C'est précisément ce à quoi sert le code spécial d'index 2^N que nous avions stocké dans le dictionnaire au début de l'algorithme (les anglais l'appellent le *clear code*) : il indique qu'il est temps de repartir avec un nouveau dictionnaire. On reprend alors simplement le dictionnaire de départ de taille 2^{N+1} et les index sont envoyés sur le flot de sortie à nouveau sur $N+1$ bits. C'est pour pouvoir se rappeler cette taille, qu'elle est stockée en début du bloc de données de l'image. Il nous reste enfin à expliquer le code d'index $2^N + 1$: celui-ci s'appelle le *end-of-information* code et sert simplement à informer LZW que l'encodage/décodage de l'image est terminé.

4.1.2 La compression conservative de JPEG

JPEG est l'acronyme de **J**oint **P**hotographic **E**xpert **G**roup. Le standard le plus connu développé à la fin des années 70 par ce groupe d'experts est le mode de compression non conservatif que nous verrons plus loin. Cependant, le groupe a aussi développé un algorithme conservatif. Celui-ci est fondé sur un codage «prédictif».

Le codage prédictif

L'idée des codages prédictifs est relativement simple : essayons de compresser la séquence d'octets suivante

$$aaaaaaaaaabbbbbbbbbbccccccccccddddddddddd$$

grâce à l'algorithme de Huffman. Tous les symboles de l'alphabet ont la même probabilité $P = 1/4$ d'apparaître. Par conséquent, Huffman va coder chaque symbole sur 2 bits, par exemple :

$$a \equiv 01 \qquad b \equiv 00 \qquad c \equiv 10 \qquad d \equiv 11,$$

et donc la séquence d'octets ci-dessus sera transformée en une séquence de 80 bits. Si, par contre, on avait eu la séquence suivante :

$$aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcd$$

Huffman aurait utilisé le code suivant :

$$a \equiv 0 \qquad b \equiv 10 \qquad c \equiv 110 \qquad d \equiv 111,$$

ce qui nous donnerait en sortie une séquence de 45 octets. Pourquoi Huffman obtient-il un meilleur taux de compression dans le deuxième cas que dans le premier ? Tout simplement parce que l'entropie est plus petite dans le deuxième cas ($H \approx 0,5$ contre $H = 2$). Cela est dû à une propriété de l'entropie que nous avions vue dans la sous-section 2.1.1, à savoir que l'entropie est maximale lorsque les probabilités d'apparition des symboles ont tendance à être uniformes. Par conséquent, si l'on voulait mieux compresser la première séquence d'octets ci-dessus, il faudrait utiliser Huffman avec des distributions de probabilité très hétérogènes. L'un des moyens pour parvenir à cela est de ne plus considérer seulement dans Huffman les probabilités d'apparition des symboles, mais leurs probabilités conditionnellement aux symboles que l'on a déjà lus sur le flot d'entrée. C'est ce que l'on appelle un algorithme prédictif car il utilise l'historique disponible pour prédire quel sera le prochain symbole lu sur le flot d'entrée.

Définition 12 (codage prédictif) : *Un codage est dit prédictif s'il utilise la connaissance des symboles précédemment lus sur le flot d'entrée pour coder le symbole actuellement lu.*

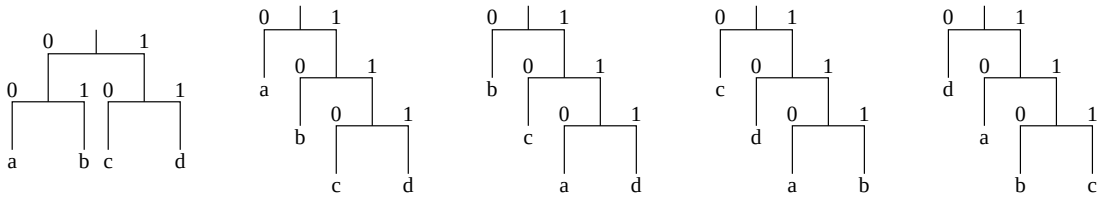
Pour revenir à la séquence d'octets :

aaaaaaaaaabbbbbbbbbbccccccccccddddddddddd

on peut établir les listes de probabilités suivantes :

$$\begin{array}{llllllllll} P(a) = \frac{1}{4} & P(b) = \frac{1}{4} & P(c) = \frac{1}{4} & P(d) = \frac{1}{4}, & & & & & & \\ P(a|a) = \frac{9}{10} & P(b|a) = \frac{1}{10} & P(c|a) = 0 & P(d|a) = 0 & P(a|b) = 0 & P(b|b) = \frac{9}{10} & P(c|b) = \frac{1}{10} & P(d|b) = 0, \\ P(a|c) = 0 & P(b|c) = 0 & P(c|c) = \frac{9}{10} & P(d|c) = \frac{1}{10} & P(a|d) = 0 & P(b|d) = 0 & P(c|d) = 0 & P(d|d) = 1, \end{array}$$

qui représentent les probabilités marginales d'apparition des symboles (les $P(a)$, $P(b)$, etc) et les probabilités d'apparition des symboles conditionnellement au dernier symbole lu sur le flot d'entrée (les $P(c|a)$, etc). Ainsi, on peut créer les arbres de Huffman suivants :



$P(\cdot)$ $P(\cdot|a)$ $P(\cdot|b)$ $P(\cdot|c)$ $P(\cdot|d)$

Par conséquent, le premier symbole lu sur le flot d'entrée est codé grâce à l'arbre de Huffman de gauche, les 10 caractères suivants sont codés grâce au deuxième arbre de Huffman, les 10 suivants grâce au troisième arbre, les 10 suivants grâce au quatrième arbre et les 9 suivants grâce au dernier arbre. On obtient ainsi la séquence de bits suivants :

00|0|0|0|0|0|0|0|0|0|10|0|0|0|0|0|0|0|0|0|10|0|0|0|0|0|0|0|0|0|10|0|0|0|0|0|0|0|0

soit une séquence de 44 bits. Le codage prédictif est donc bien meilleur que l'algorithme classique de Huffman (80 bits).

Le standard actuel JPEG

Revenons à JPEG conservatif. Celui-ci utilise donc un codage prédictif. Plus exactement, le standard actuel procède de la manière suivante (cf. G.K. Wallace, «The JPEG Still Picture Compression Standard», *Communications of the ACM*, vol 34, pp.31–44, avril 1991) : Un prédicteur combine les valeurs de 0 à 3 voisins du point X que l'on veut coder, pour former une prédiction $\widehat{X}$ de la valeur que devrait avoir ce point. L'encodeur calcule alors la différence $X - \widehat{X}$ et encode celle-ci grâce à un algorithme statistique (Huffman ou codage arithmétique). JPEG est muni de huit prédicteurs :

1. pas de prédiction,
2. $\widehat{I}(i,j) = I(i-1,j)$,
3. $\widehat{I}(i,j) = I(i,j-1)$,
4. $\widehat{I}(i,j) = I(i-1,j-1)$,
5. $\widehat{I}(i,j) = I(i,j-1) + I(i-1,j) - I(i-1,j-1)$,
6. $\widehat{I}(i,j) = I(i,j-1) + [I(i-1,j) - I(i-1,j-1)]/2$,
7. $\widehat{I}(i,j) = I(i-1,j) + [I(i,j-1) - I(i-1,j-1)]/2$,
8. $\widehat{I}(i,j) = [I(i,j-1) + I(i-1,j)]/2$,

où $I(i,j)$ représente le pixel lu à la ligne i et la colonne j, et où $\widehat{I}(i,j)$ représente la valeur estimée par le prédicteur pour le pixel de coordonnées (i,j). Généralement, le codage ci-dessus permet d'obtenir des taux de compression de l'ordre de 0,5 (autrement dit, on gagne un facteur 2 sur un disque dur). Pour donner une idée, voici un tableau de comparaison entre les tailles d'images JPEG conservatif (le meilleur parmi les 8 modes ci-dessus, encodé avec un code arithmétique adaptatif) et GIF (source : K. Sayood, «Introduction to Data Compression», second edition, 1996, *Academic Press*) :

Image	Meilleur JPEG	GIF
Sena	31055 octets	51085 octets
Sensin	32429 octets	60649 octets
Earth	32137 octets	34276 octets
Omaha	48818 octets	61341 octets

Le nouveau standard JPEG : JPEG-LS

Il existe maintenant un nouveau standard JPEG conservatif : il s'agit de JPEG-LS, dont l'algorithme est aussi connu sous le nom de LOCO-I (acronyme de «LOw COmplexity LOssless COmpression for Images», cf. M. Weinberger, G. Seroussi et G. Sapiro, «The LOCO-I Lossless Image Compression Algorithm : Principles and Standardization into JPEG-LS», *Hewlett-Packard Laboratories Technical Report* No. HPL-98-193R1, Novembre 1998). Cet algorithme est composé de trois phases :

1. un prédicteur estime la valeur la plus probable $\widehat{x}_{i+1}$ du $i+1^{\text{ème}}$ pixel encodé, à partir d'un historique des pixels précédemment lus sur le flot d'entrée ;
2. une détermination du contexte dans lequel x_{i+1} apparaît, c'est-à-dire une fonction agrégeant les informations sur les derniers pixels lus sur le flot d'entrée ;
3. l'encodage des erreurs résiduelles $e_{i+1} = x_{i+1} - \widehat{x}_{i+1}$ grâce à un code de Golomb-Rice (cf. la sous-section 2.4.2).

Le choix des algorithmes utilisés pour chacune des phases est largement conditionné par le fait que l'on veut obtenir un standard ayant une complexité de calculs très peu élevée.

Afin de réduire la complexité des calculs, la phase de prédiction utilise un historique très restreint : seuls les pixels a, b et c de la figure 11 sont utilisés pour prédire la valeur du pixel x. Le prédicteur est décrit par la

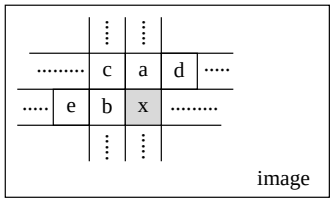


FIG. 11 – Les pixels utilisés pour prédire x dans JPEG-LS.

formule suivante :

$$\widehat{x}_{i+1} = \begin{cases} \min(a,b) & \text{si } c \geq max(a,b), \\ \max(a,b) & \text{si } c \leq min(a,b), \\ a+b-c & \text{sinon.} \end{cases}$$

À première vue, cette formule peut paraître compliquée pour un algorithme qui se veut le plus rapide possible. En fait, c'est que le prédicteur intègre un détecteur (primitif) de contours. En effet, lorsque $c \geq max(a,b)$ (resp. $c \leq min(a,b)$), cela dénote souvent la présence d'un contour vertical (resp. horizontal) à gauche (resp. au dessus) du pixel x. Dans ce cas, le prédicteur estime qu'il y a de bonnes chances que le pixel x appartienne au même objet que a (resp. b), c'est pourquoi $\widehat{x}_{i+1}$ est estimé par a (resp. b). Lorsqu'aucun contour n'est détecté (le troisième cas du prédicteur), on suppose que les pixels a, b, c et x font partie du même objet et x est estimé grâce à une sorte de médiane entre les trois points, la médiane est toutefois asymmétrique car c est plus éloigné de x que les pixels a et b.

Afin de coder l'erreur d'estimation avec un algorithme à la Huffman, on a besoin de connaître la probabilité d'apparition de l'erreur. Comme on l'avait vu précédemment avec le codage prédictif, on a intérêt à encoder la probabilité d'erreur conditionnellement à un historique : le contexte. C'est pourquoi, si l'on veut un codage efficace, on doit avoir un bon algorithme de détermination du contexte. Dans LOCO-I, celui-ci est construit à partir de trois différences :

$$g_1 = d - b \qquad g_2 = b - c \qquad g_3 = c - a.$$

Ces différences capturent le niveau d'activité (variations brusques ou légères de couleurs) localement autour du pixel x. Elles gouvernent le comportement statistique des erreurs de prédiction dans la mesure où, pour encoder Huffman, on va utiliser comme loi de probabilité :

$$P(e_{i+1} = \Delta | g_1, g_2, g_3).$$

En fait, LOCO-I n'utilise pas tout à fait cette distribution de probabilité car cela demanderait trop de capacité mémoire pour être opérationnel. En effet, si les valeurs des pixels varient entre 0 et 255, il y a $256^3 = 16777216$ triplets (g_1, g_2, g_3) possibles. Pour simplifier, donc, LOCO-I utilise une fonction $k(\cdot)$, que les anglais appellent *quantization*, et qui associe à chaque valeur g_i un nombre Q_i dans l'ensemble $\{-4,-3,-2,-1,0,1,2,3,4\}$:

$$\begin{aligned}
g_i \leq -T_3 &\Rightarrow Q_i = -4,\\
-T_3 < g_i \leq -T_2 &\Rightarrow Q_i = -3,\\
-T_2 < g_i \leq -T_1 &\Rightarrow Q_i = -2,\\
-T_1 < g_i \leq 0 &\Rightarrow Q_i = -1,\\
g_i = 0 &\Rightarrow Q_i = 0,\\
0 < g_i \leq T_1 &\Rightarrow Q_i = 1,\\
T_1 < g_i \leq T_2 &\Rightarrow Q_i = 2,\\
T_2 < g_i \leq T_3 &\Rightarrow Q_i = 3,\\
T_3 < g_i &\Rightarrow Q_i = 4.
\end{aligned}$$

Les valeurs T_i sont calculées de manière à ce que les Q_i forment 9 classes équiprobables. Ainsi, le nombre de triplets passe de 16777216 à $9 \times 9 \times 9 = 729$. On obtient donc 729 contextes différents. Or, en remarquant que $k(-g_i) = -k(g_i)$, et en supposant (ce qui est assez logique) que :

$$P(e_{i+1} = \Delta | Q_1, Q_2, Q_3) = P(e_{i+1} = -\Delta | -Q_1, -Q_2, -Q_3),$$

on peut encore réduire l'utilisation mémoire en ne stockant que la moitié de la distribution de probabilité conditionnelle P. Pour une valeur donnée de Δ, on a donc en principe que $((2 \times 4 + 1)^3 + 1)/2 = 365$ valeurs à stocker dans la table de probabilité.

Maintenant que nous avons les probabilités d'apparition des erreurs d'estimation e_{i+1}, il suffit d'utiliser un codage statistique pour finir la compression. On pourrait utiliser Huffman, mais c'est en fait une variante qui a été choisie : un code de Golomb. Pourquoi utiliser un tel code dans LOCO-I? Eh bien, nous avions vu que le code de Golomb est optimal lorsque la distribution de probabilité des entiers (cf. figure 8 page 45) est définie par :

$$P(n) = p^{n-1}(1-p), \text{ pour un } p \in]0,1[$$

et lorsque m vérifie :

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil.$$

Or, il s'avère que les erreurs d'estimation e_{i+1} de LOCO-I suivent une distribution similaire au $P(\cdot)$ ci-dessus (en fait, c'est très légèrement différent car il faut considérer les erreurs positives et les erreurs négatives, cf. la figure 12). Voyons maintenant les résultats pratiques obtenus par les auteurs de LOCO-I :

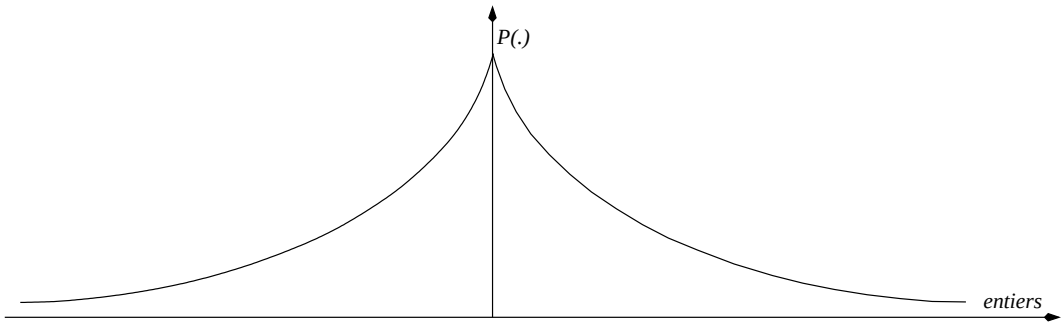


FIG. 12 – La distribution de probabilité des erreurs d'estimation.

Image	LOCO-I	JPEG standard Huffman	JPEG standard arithmétique
bike	3,59	4,34	3,92
cafe	4,80	5,74	5,35
woman	4,17	4,86	4,47
tools	5,07	5,71	5,47
bike3	4,37	5,18	4,78
cats	2,59	3,73	2,74
us	2,67	3,77	2,52

TAB. 15: nombre de bits/pixel obtenus avec LOCO-I et le standard actuel JPEG.

4.2 Préliminaires mathématiques à la compression non conservative

Les images possèdent une propriété très importante pour la compression : elles peuvent être légèrement modifiées pendant la compression/décompression sans affecter la qualité perçue par l'utilisateur. C'est cette propriété qui est le fondement de la compression non conservative. Oui, mais problème : comment réaliser de «légères modifications» de l'image ? Plusieurs techniques sont possibles. Nous allons voir ici ce que JPEG utilise, à savoir la quantification de transformées par DCT (**D**iscrete **C**osine **T**ransform). La suite de cette sous-section est fortement inspirée du livre de Mark Nelson, «The Data Compression Book», $2^{\text{ème}}$ édition (attention : dans ce livre, la formule générale de la DCT est fausse. En fait elle n'est valide que pour $N = 8$).

4.2.1 Qu'est ce que la DCT ?

La DCT est une classe particulière de transformée de Fourrier (on parle aussi de méthode spectrale). L'idée de ces transformées est de prendre un signal et de le transformer d'un type de représentation à un autre. Le plus simple, pour introduire la transformée de Fourrier est de partir du domaine de la physique : un processus physique peut être décrit soit comme une quantité h variant dans le temps — donc une fonction $h(t)$ — soit comme une amplitude H fonction de la fréquence f du phénomène physique. Généralement, H est un nombre complexe indiquant grâce à une partie imaginaire la phase dans laquelle on se trouve. Ainsi, si t est mesuré en secondes, f est, lui, mesuré en cycles par seconde ou Hertz. Si $h(\cdot)$ est exprimé en mètres (h peut représenter une position sur un axe), alors $H(\cdot)$ est une fonction en cycles par mètre. Évidemment, bien que ces deux représentations soient différentes, elles représentent le même phénomène. Il devrait donc être possible de passer de l'une à l'autre. Ce passage s'appelle la transformée de Fourrier :

Définition 13 (transformée de Fourrier) : *Soit un phénomène physique représenté par une fonction h dépendant du temps t, mais aussi par une fonction H dépendant de la fréquence f. Alors on peut passer d'une représentation à l'autre grâce aux équations de la transformée de Fourrier :*

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt,$$
$$h(t) = \int_{-\infty}^{\infty} H(f) e^{2\pi i f t} df.$$

Cette transformée est très utilisée pendant l'analyse d'échantillons audio. Par exemple, lorsque l'on collecte des échantillons à partir d'un signal audio, on récupère une courbe temporelle du signal (cf. la figure 13). Ce signal est composé de trois courbes sinusoïdales ajoutées ensemble. Les abscisses représentent différents points

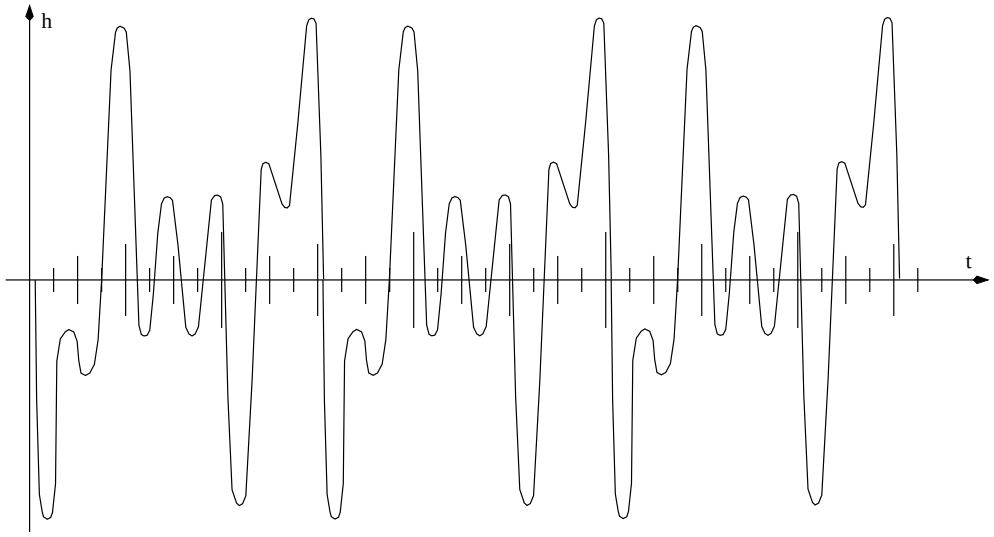


FIG. 13 – Une représentation temporelle $h(t)$ d'un signal audio.

dans le temps et les ordonnées l'amplitude du signal. Lorsqu'on applique la transformée de Fourrier, on obtient alors la représentation de la figure 14. Maintenant, les abscisses représentent des fréquences et les ordonnées

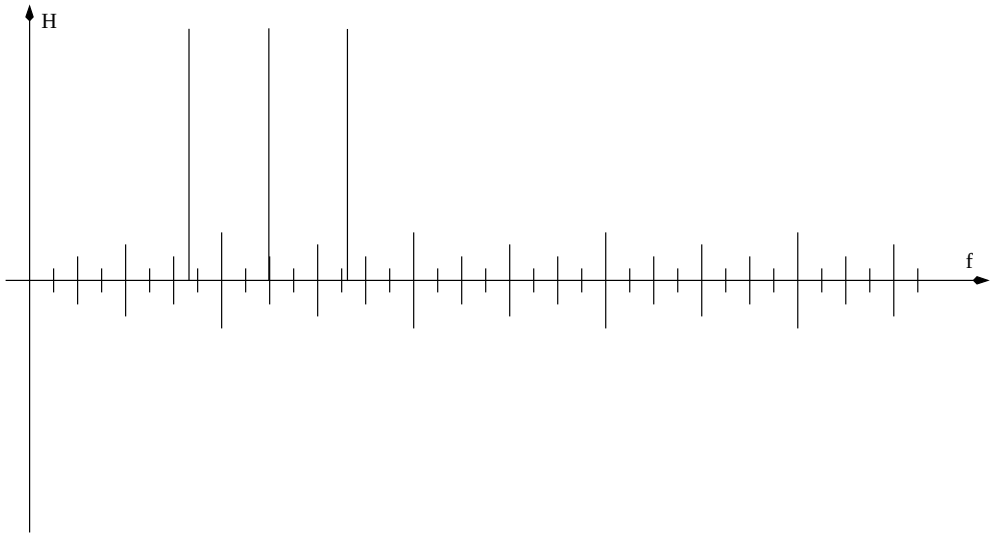


FIG. 14 – Une représentation fréquentielle $H(f)$ du signal audio.

représentent la grandeur de chaque sinusoïde. La signification de la transformée de Fourrier est alors évidente : elle indique simplement que le signal est composé par la somme de 3 fréquences différentes de tailles sensiblement identiques.

On voit que la deuxième représentation est bien plus économique que la figure 13. Pourtant, les deux courbes représentent le même signal. Ceci vient du fait que le signal n'est pas extrêmement complexe : il est composé de très peu de fréquences. Or la connaissance de ces quelques fréquences suffit à reconstituer tout le signal. D'après les formules de Fourrier, la donnée d'une des représentations nous permet de calculer la deuxième et inversement. Par conséquent, on peut déjà réaliser une certaine compression du signal juste en sélectionnant celle des deux courbes qui est la plus économique à représenter.

La DCT est relativement similaire à la transformée de Fourrier décrite ci-dessus. Simplement, elle prend un ensemble de points d'un domaine spatial et les transforme en une représentation identique dans un domaine de fréquence. Comme le domaine est spatial et non plus temporel (c'est une image, quoi), la fonction h ne dépend plus d'un seul argument, mais de deux : les axes X et Y usuels. La fonction h, en ce qui nous concerne, sera simplement la valeur d'un pixel en un point particulier de l'image. La fonction H est alors une représentation «spectrale» de l'image. C'est aussi une fonction à deux dimensions, représentant respectivement les fréquences du signal sur l'axe des X et sur l'axe des Y. Comme pour la transformée de Fourrier, on peut passer d'une courbe à l'autre et réciproquement. La formule de la DCT, qui découle de celle de la transformée de Fourrier, est la suivante :

Définition 14 (Discrete Cosine Transform) : *Soit une image carrée de N lignes et N colonnes. Appelons pixel(x,y) la valeur du pixel aux coordonnées (x,y) de l'image. Alors :*

$$DCT(i,j) = C(i)C(j)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} pixel(x,y)\cos\left(\frac{(2x+1)i\pi}{2N}\right)\cos\left(\frac{(2y+1)j\pi}{2N}\right),$$

$$pixel(x,y) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} C(i)C(j)DCT(i,j)\cos\left(\frac{(2x+1)i\pi}{2N}\right)\cos\left(\frac{(2y+1)j\pi}{2N}\right),$$

où $C(n) = 1/\sqrt{2N}$ si $n = 0$ et $C(n) = 1/\sqrt{N}$ sinon.

En fait, lorsqu'on programme les équations ci-dessus, pour des raisons d'efficacité, on stocke tous les cosinus dans une table `Cos`. Les calculs se font alors grâce à un programme très simple :

```
for (i=0, nb1 = 2.0 / N, nb2 = 1.0 / N, nb3 = sqrt(2.0) / N; i<N; i++)
  for (j=0; j<N; j++)
    {
      tmp = 0.0;
      for (x = 0; x<N; x++)
        for (y = 0; y<N; y++)
          tmp += Cos[x][i] * Cos[y][i] * pixel[x][y];

      if (i != 0) && (j != 0)) Cij = nb1;
      else if ((i == 0) && (j == 0)) Cij = nb2;
      else Cij = nb3;
      DCT[i][j] = tmp * Cij;
    }
```

Tout comme la tranformée de Fourrier avait permis de représenter toutes les informations de la courbe 13 en utilisant seulement 3 «bâtons», la DCT permet de représenter des images complexes avec relativement peu de données. En effet, dans la matrice $DCT(i,j)$, tous les éléments sur la ligne 0 ont une composante en fréquence nulle sur une direction du signal, et tous les éléments sur la colonne 0 ont une composante en fréquence nulle sur l'autre direction. Et, plus un index i ou j est grand, plus la fréquence correspondante est élevée, les plus hautes fréquences correspondant aux index $N-1$ de la matrice. Or, il s'avère que les images de nos écrans d'ordinateurs sont principalement composées de basses fréquences. Donc plus les index augmentent, plus les fréquences augmentent, moins l'information est importante pour la description de l'image et moins la $DCT(i,j)$ est grande. Par conséquent, on peut dire que la DCT identifie les parties importantes de l'image (celles d'index petits) et les parties qui peuvent être supprimées sans affecter la qualité de l'image (les grands index). avant de voir comment JPEG utilise ces informations pour obtenir les taux de compression énormes qu'on lui connaît, voyons comment améliorer un peu le calcul de la DCT.

4.2.2 Amélioration du calcul de la DCT

Le petit programme de calcul de la matrice $DCT_{0\leq i,j<N}$ écrit ci-dessus a une complexité vraiment très mauvaise. En effet, pour chaque valeur de (i,j), on doit effectuer un algorithme en $O(N^2)$ pour calculer $DCT(i,j)$ puisque l'on a deux boucles imbriquées en `x` et en `y`. Comme il y a N^2 valeurs possibles pour les couples (i,j),

on se retrouve globalement avec un algorithme en $O(N^4)$ pour calculer $DCT_{0\leq i,j<N}$. Lorsque la taille de l'image augmente, le calcul de cette matrice augmente donc très vite et rend ainsi l'algorithme inutilisable en pratique sur la totalité de l'image : un utilisateur n'est pas prêt à attendre 5 minutes pour la voir. Pour un (i,j) donné, on ne peut pas vraiment réduire complexité du calcul de $DCT(i,j)$, c'est pourquoi JPEG découpe l'image à compresser en petits blocs et calcule $DCT_{0\leq i,j<N}$ sur chacun de ces blocs séparément. Bien évidemment, plus le bloc est grand, plus il permet d'obtenir des taux de compression appréciables, mais plus la DCT est longue à calculer. JPEG a donc choisi des blocs de taille de 8×8 pixels.

En plus de la réduction en blocs, on peut encore réduire la complexité globale du calcul de toute la matrice $DCT_{0\leq i,j<N}$. En effet, la formule de la DCT avec ses sommes fait furieusement penser à des produits matrice-vecteur. Ainsi, si

$$COS(i,j) = \begin{cases} \dfrac{1}{\sqrt{N}} & \text{si } i = 0, \\ \sqrt{\dfrac{2}{N}} \cos\left(\dfrac{(2j+1)i\pi}{2N}\right) & \text{si } i > 0, \end{cases}$$

alors la matrice $DCT_{0\leq i,j<N}$ peut s'exprimer de la façon suivante :

$$DCT = COS_{0\leq i,j<N} \otimes pixel_{0\leq i,j<N} \otimes COS^T_{0\leq i,j<N},$$

où $COS^T_{0\leq i,j<N}$ représente la transposée de $COS_{0\leq i,j<N}$ et où $\otimes$ représente l'opération produit de deux matrices. Cela nous amène donc à considérer le code suivant pour construire la matrice $DCT_{0\leq i,j<N}$.

```
void calcule_DCT(float **pixel, float **DCT, int N)
{
  int     i,j,k;
  float   sqr = 1.0 / sqrt((float) N);
  float **COS, **temp;

  /* creation et initialisation de la matrice COS */
  if ((COS = (float **) malloc(N * sizeof(float *))) == NULL) exit(1);
  if ((COS[0] = (float *) malloc(N * sizeof(float))) == NULL) exit(1);
  for (j = 0; j<N; j++) COS[0][j] = sqr;
  sqr = sqrt(2.0 / N);
  for (i = 1; i<N; i++)
    {
      if ((COS[i] = (float *) malloc(N * sizeof(float))) == NULL) exit(1);
      for (j=0; j<N; j++)
        COS[i][j] = sqr * cos((2*j+1) * i * M_PI / (2.0 * N));
    }

  /* calcul du produit temp = pixel * COS^T */
  if ((temp = (float **) malloc(N * sizeof(float *))) == NULL) exit(1);
  for (i=0; i<N; i++)
    {
      if ((temp[i] = (float *) malloc(N * sizeof(float))) == NULL) exit(1);
      for (j=0; j<N; j++)
        for (k=0, temp[i][j] = 0.0; k<N; k++)
          temp[i][j] += pixel[i][k] * COS[j][k];
    }

  /* calcul de COS * (pixel * COS^T) */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      for (k=0, DCT[i][j] = 0.0; k<N; k++)
        DCT[i][j] += COS[i][k] * temp[k][j];
```

```
  /* desallocation de temp et de COS */
  for (i=0; i<N; i++)
    {
      free((char *) COS[i]);
      free((char *) temp[i]);
    }
  free((char *)COS);
  free((char *)temp);
}
```

On voit dans l'algorithme ci-dessus qu'il n'y a plus que trois boucles imbriquées, d'où une complexité en $O(N^3)$. Ce n'est pas encore génial mais cela permet d'aller tout de même beaucoup plus vite que le précédent code. Si l'on veut maintenant améliorer la complexité pratique de l'algorithme, il faut se débarrasser de l'arithmétique en virgule flottante et passer à une arithmétique entière, exactement comme nous l'avions fait pour le codage arithmétique. On peut maintenant voir ce que cela donne sur un exemple : si l'on applique l'algorithme ci-dessus au tableau suivant :

163	166	177	175	173	175	166	132	129	121	122	127
170	139	143	166	120	126	172	136	131	128	147	129
122	170	122	163	127	175	149	162	162	138	149	147
133	136	151	142	129	153	141	179	164	157	158	164
155	150	173	166	151	140	176	153	122	170	136	121
165	165	164	147	176	125	166	129	133	137	144	135
162	165	134	147	134	164	123	170	127	168	156	158
128	144	123	131	134	139	132	120	176	168	139	172
166	126	174	179	135	130	134	178	167	141	137	122
177	132	172	176	121	140	146	121	165	149	124	179
160	129	171	157	177	131	141	155	129	127	147	136
137	153	126	125	166	143	179	164	148	171	152	149

TAB. 16: Matrice des teintes de gris d'une image 12×12

on obtient la matrice 12×12 suivante :

1783.58	28.54	-4.09	7.90	-9.08	-14.04	7.42	23.81	18.80	-8.59	2.50	-5.01
0.03	27.63	-18.87	-3.80	14.42	-10.41	-26.60	5.12	-4.81	-0.89	-28.45	-15.95
0.55	22.15	-26.96	-10.27	16.17	-8.10	4.56	11.30	3.63	3.47	-10.64	17.32
-12.87	27.25	19.62	-6.35	-8.97	3.67	22.00	-9.65	34.63	15.69	-19.81	-5.62
5.00	31.25	-5.47	-12.94	31.50	26.58	-27.25	0.47	-36.08	-30.98	22.09	20.20
15.27	79.61	-0.34	-23.79	-12.59	-10.53	6.05	31.70	30.36	9.63	28.43	34.12
8.75	-19.41	-26.20	11.13	-17.66	1.70	-8.08	-11.35	-0.53	-4.70	15.38	3.17
6.28	-20.34	8.51	-21.74	8.06	3.90	3.76	-33.86	-4.37	18.13	9.23	-25.49
26.22	-3.59	-5.24	5.52	5.05	-6.49	4.54	-23.10	-6.08	10.41	-7.83	-21.53
-3.42	21.67	-25.07	-16.98	-9.65	2.70	5.41	-17.28	-28.95	4.85	-5.94	22.04
4.80	-10.07	1.05	-10.86	9.64	9.21	-18.85	-34.35	25.56	-33.00	-20.46	-6.69
13.98	26.68	-23.19	6.31	-8.64	-1.34	-4.82	22.69	-21.61	14.97	10.34	-38.24

TAB. 17: Matrice DCT résultante

On voit très bien sur cette matrice ce qui avait été annoncé dans la sous-section précédente, à savoir que plus on s'éloigne du coin supérieur gauche de la matrice, plus les nombres sont petits. Pour vous convaincre que les fréquences les plus importantes pour la qualité de l'image sont ceux en haut à gauche, j'ai délibérément mis à 0 tous les éléments de la DCT qui se trouvent en dessous de la deuxième diagonale de la matrice :

1783.58	28.54	-4.09	7.90	-9.08	-14.04	7.42	23.81	18.80	-8.59	2.50	-5.01
0.03	27.63	-18.87	-3.80	14.42	-10.41	-26.60	5.12	-4.81	-0.89	-28.45	0.00
0.55	22.15	-26.96	-10.27	16.17	-8.10	4.56	11.30	3.63	3.47	0.00	0.00
-12.87	27.25	19.62	-6.35	-8.97	3.67	22.00	-9.65	34.63	0.00	0.00	0.00
5.00	31.25	-5.47	-12.94	31.50	26.58	-27.25	0.47	0.00	0.00	0.00	0.00
15.27	79.61	-0.34	-23.79	-12.59	-10.53	6.05	0.00	0.00	0.00	0.00	0.00
8.75	-19.41	-26.20	11.13	-17.66	1.70	0.00	0.00	0.00	0.00	0.00	0.00
6.28	-20.34	8.51	-21.74	8.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00
26.22	-3.59	-5.24	5.52	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
-3.42	21.67	-25.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4.80	-10.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
13.98	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

TAB. 18: Matrice DCT modifiée

Le calcul de la DCT inverse nous donne la matrice image suivante :

167.43	162.25	171.61	185.84	161.31	174.46	169.61	132.70	128.75	122.75	123.06	126.22
158.23	151.80	148.16	153.96	130.92	144.17	153.71	135.50	132.41	128.40	134.34	135.42
138.41	142.92	139.02	147.79	134.89	146.54	167.89	165.92	155.21	147.02	153.49	146.90
128.14	145.24	144.29	151.52	140.12	140.74	161.13	168.26	157.52	155.16	163.76	151.13
146.07	164.02	161.02	161.60	150.77	145.57	155.20	155.44	144.70	147.36	149.52	131.74
167.04	170.54	161.86	158.11	154.26	155.54	153.36	138.37	130.10	138.71	134.89	123.21
163.28	149.97	143.42	140.83	139.72	147.32	144.60	134.05	145.20	165.83	163.16	170.64
141.99	124.19	135.07	139.67	130.76	132.25	136.42	142.82	160.94	164.83	141.19	155.86
158.79	140.86	162.60	166.34	142.60	128.54	138.55	159.80	171.18	156.45	121.56	141.72
165.40	148.57	169.64	171.08	147.61	125.22	131.86	149.99	149.66	145.74	133.63	163.60
156.36	142.56	157.60	161.03	158.04	144.33	145.55	146.77	128.16	136.32	134.59	148.69
146.86	132.08	135.71	136.23	152.00	156.32	167.12	169.39	149.18	166.43	157.81	143.87

TAB. 19: Matrice image inverse de la DCT modifiée

Notez que la matrice image obtenue est relativement proche des teintes de gris de l'image d'origine. Pour vous aider à faire cette comparaison, j'ai représenté dans le tableau ci-dessous la déviation relative des pixels de l'image finale par rapport aux mêmes pixels de l'image d'origine. La formule utilisée a été la suivante :

$$\text{déviation}(i,j) = \frac{\text{pixel}(i,j) \text{ de l'image modifiée} - \text{pixel}(i,j) \text{ de l'image d'origine}}{\text{pixel}(i,j) \text{ de l'image d'origine}}.$$

-0.03	0.02	0.03	-0.06	0.07	0.00	-0.02	-0.01	0.00	-0.01	-0.01	0.01
0.07	-0.08	-0.03	0.08	-0.08	-0.13	0.12	0.00	-0.01	0.00	0.09	-0.05
-0.12	0.19	-0.12	0.10	-0.06	0.19	-0.11	-0.02	0.04	-0.06	-0.03	0.00
0.04	-0.06	0.05	-0.06	-0.08	0.09	-0.12	0.06	0.04	0.01	-0.04	0.09
0.06	-0.09	0.07	0.03	0.00	-0.04	0.13	-0.02	-0.16	0.15	-0.09	-0.08
-0.01	-0.03	0.01	-0.07	0.14	-0.20	0.08	-0.07	0.02	-0.01	0.07	0.10
-0.01	0.10	-0.07	0.04	-0.04	0.11	-0.15	0.27	-0.13	0.01	-0.04	-0.07
-0.10	0.16	-0.09	-0.06	0.02	0.05	-0.03	-0.16	0.09	0.02	-0.02	0.10
0.05	-0.11	0.07	0.08	-0.05	0.01	-0.03	0.11	-0.02	-0.10	0.13	-0.14
0.07	-0.11	0.01	0.03	-0.18	0.12	0.11	-0.19	0.10	0.02	-0.07	0.09
0.02	-0.10	0.09	-0.03	0.12	-0.09	-0.03	0.06	0.01	-0.07	0.09	-0.09
-0.07	0.16	-0.07	-0.08	0.09	-0.09	0.07	-0.03	-0.01	0.03	-0.04	0.04

TAB. 20: déviation relative de l'image modifiée

4.3 La compression JPEG

Le Joint Photographic Expert Group a produit en 1991 une proposition préliminaire de normalisation pour la compression de données graphique. En ont résultées les fameuses images JPEG. Grossièrement, l'algorithme de compression peut être résumé sur la figure suivante :

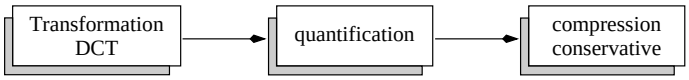


FIG. 15: L'algorithme de compression non conservative de JPEG.

4.3.1 La quantification

Nous avons vu dans la section précédente comment effectuer la première étape, le calcul de la DCT. Mais la DCT, en elle-même, ne réalise aucune compression : on part d'une matrice de $N \times N$ pixels et on obtient une nouvelle matrice de taille $N \times N$. Ce qui réalise la compression non conservative, c'est l'étape 2 : la quantification. Elle consiste à réduire le nombre de bits nécessaires au stockage de la matrice DCT. Le procédé est simple : on va diviser tous les éléments de la DCT par une certaine quantité (le quantum) et on ne conservera que la partie entière au plus près :

$$\boxed{\text{valeur quantifiée}(i,j) = \text{arrondi à l'entier le plus proche } \left(\frac{DCT(i,j)}{\text{Quantum}(i,j)}\right)}.$$

À l'instar de ce que nous avons vu à la fin de la sous-section précédente, cette manipulation va détériorer légèrement l'image, mais si l'on choisit des quanta corrects, cette détérioration sera peu ou pas visible. Comme les éléments de la matrice DCT sont de moins en moins importants pour l'image lorsque l'on se déplace vers la droite et vers le bas de la matrice, on pourra affecter de forts quanta sur ces éléments et de petits quanta pour les éléments les plus importants de l'image, à savoir ceux en haut à gauche de la DCT. Lorsque l'on voudra décompresser l'image JPEG, il suffira d'effectuer la transformation inverse :

$$\boxed{DCT(i,j) = \text{valeur quantifiée}(i,j) \times \text{Quantum}(i,j)}.$$

Il reste maintenant à choisir la matrice $\text{Quantum}_{0 \leq i,j < N}$. En théorie la norme JPEG supporte n'importe quelle matrice $\text{Quantum}_{0 \leq i,j < N}$. Cela dit, l'ISO, l'International Standards Organization, a développé un ensemble standard de valeurs de quantification fournies aux programmeurs de codes JPEG. Ces tables reposent sur des tests intensifs réalisés par les membres du comité JPEG et elles fournissent une bonne base pour établir des niveaux de compression. L'une de ces tables est la suivante :

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

TAB. 21: table des quanta fournie par l'ISO

Rappelons que JPEG n'utilise que des DCT de taille 8×8. C'est pourquoi la taille de la matrice ci-dessus est de 8×8. Si on l'utilise sur l'image 8×8 suivante, qui contient des intensité en teintes de gris :

163	166	177	175	173	175	166	132
129	121	122	127	170	139	143	166
120	126	172	136	131	128	147	129
122	170	122	163	127	175	149	162
162	138	149	147	133	136	151	142
129	153	141	179	164	157	158	164
155	150	173	166	151	140	176	153
122	170	136	121	165	165	164	147

on obtient la DCT suivante :

1197.50	-27.20	-18.62	-7.61	-16.25	0.48	-15.56	-0.70
-12.07	6.26	-11.62	3.81	8.62	4.23	16.57	4.90
30.78	0.59	-13.79	18.75	-8.92	-14.79	-5.57	29.65
49.11	17.49	-11.86	3.50	22.61	21.26	-1.43	-4.52
18.75	13.85	6.91	24.19	-22.00	-10.35	-4.22	-25.49
38.27	17.76	-11.72	-15.15	-20.75	20.68	-1.48	-20.71
3.76	7.14	-23.07	-5.82	-19.58	-1.89	-0.71	-1.54
-9.32	38.49	5.09	-12.03	-9.94	37.56	17.18	40.55

Une fois appliquée la quantification avec la table fournie par l'ISO, on obtient :

75	-2	-2	0	-1	0	0	0
-1	1	-1	0	0	0	0	0
2	0	-1	1	0	0	0	1
4	1	-1	0	0	0	0	0
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Après avoir appliqué l'opération de déquantification, on obtient :

1200	-22	-20	0	-24	0	0	0
-12	12	-14	0	0	0	0	0
28	0	-16	24	0	0	0	56
56	17	-22	0	0	0	0	0
18	22	0	0	0	0	0	0
48	35	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

D'où la matrice image correspondante :

168.86	166.02	188.14	165.21	192.32	164.90	163.70	132.36
116.79	122.34	137.31	131.61	150.40	149.23	153.63	139.71
123.60	136.47	132.10	137.15	127.43	146.04	143.88	146.32
146.42	165.63	148.59	160.32	126.64	153.11	139.95	149.90
128.96	152.38	142.01	160.27	130.89	158.76	144.99	154.02
137.25	153.62	154.09	162.30	151.96	166.26	158.38	156.85
157.79	159.28	167.62	154.95	168.34	164.17	167.57	153.57
139.69	131.56	147.68	124.27	160.22	149.64	166.95	147.66

et une déviation relative de l'image :

-0.03	0.00	-0.06	0.06	-0.10	0.06	0.01	0.00
0.10	-0.01	-0.11	-0.04	0.13	-0.07	-0.07	0.19
-0.03	-0.08	0.30	-0.01	0.03	-0.12	0.02	-0.12
-0.17	0.03	-0.18	0.02	0.00	0.14	0.06	0.08
0.26	-0.09	0.05	-0.08	0.02	-0.14	0.04	-0.08
-0.06	0.00	-0.08	0.10	0.08	-0.06	0.00	0.05
-0.02	-0.06	0.03	0.07	-0.10	-0.15	0.05	0.00
-0.13	0.29	-0.08	-0.03	0.03	0.10	-0.02	0.00

On voit donc que l'application de la phase de quantification n'a pas énormément dégradé l'image. Cela dit, JPEG peut aussi utiliser une autre approche, qui consiste à sélectionner soi-même une matrice de quantification relativement simple, qui garantit que les éléments les plus élevés de cette matrice seront en bas à droite. La formule en est la suivante :

$$\boxed{\text{Quantum}(i,j) = 1 + (i+j) \times \text{Qualité}},$$

où Qualité est un nombre strictement positif. Plus ce nombre est proche de 1, plus la qualité est bonne, mais moins le taux de compression sera élevé. Pour donner une idée, j'ai calculé la DCT d'une image, je l'ai quantifiée

pour des valeurs de Qualité variant de 1 à 25, j'ai déquantifié la DCT, puis recalculé l'image correspondante. J'ai alors calculé la déviation relative de la nouvelle image, puis l'écart-type de cette déviation. Le tableau ci-dessous rassemble ces écart-types. Rappelons que l'on a très peu de chances de se trouver à plus de 3 écart-types du pixel cherché.

qualité	écart-type	qualité	écart-type	qualité	écart-type	qualité	écart-type	qualité	écart-type
1	0.016086	6	0.098209	11	0.107455	16	0.113450	21	0.118826
2	0.031841	7	0.101824	12	0.109077	17	0.115793	22	0.121171
3	0.051360	8	0.106092	13	0.111452	18	0.115319	23	0.121509
4	0.065297	9	0.105702	14	0.111254	19	0.117978	24	0.122431
5	0.087912	10	0.106116	15	0.112454	20	0.119366	25	0.121495

4.3.2 La phase de compression conservative

Après avoir obtenu une DCT quantifiée, JPEG compresse celle-ci en utilisant un algorithme conservatif. Cet algorithme combine trois étapes :

1. la première modifie le coefficient d'index $(0,0)$ — l'élément en haut à gauche — des matrices DCT d'une valeur absolue à une valeur relative. Autrement dit, elle calcule la différence entre cet élément de la DCT actuelle et celui de la DCT précédente. Comme les blocs adjacents d'une image ont, en principe, une forte corrélation, à l'issue de cette étape, on obtient généralement des nombres très petits ;
2. dans une deuxième étape, les coefficients de la DCT sont réordonnés suivant une séquence en zigzag ;
3. enfin, ils sont codés en utilisant du RLE pour les coefficients nuls, puis le tout est encodé grâce à une méthode statistique (Huffman ou codage arithmétique).

Étudions maintenant en détails chacune des étapes. La première se contente d'effectuer une différence entre l'élément en haut à gauche de la DCT actuelle et celui de la DCT précédente. Pour la première DCT, on se contente de garder la valeur de l'élément en haut à gauche de la DCT. Par exemple, si les trois premières DCT ont respectivement pour élément d'index $(0,0)$ les nombres 1197, 1190 et 1193, alors la première DCT est encodée inchangée, la deuxième DCT est encodée avec pour élément d'index $(0,0)$ le nombre $1190-1197=-7$, et la troisième DCT avec l'élément d'index $(0,0)$ le nombre $1193-1190=3$. Certes, me direz-vous, mais comment vais-je coder ces différences pour économiser le plus de bits en sortie ? Hein, comment ? Eh bien, tout simplement en utilisant du Huffman et en remarquant que plus les nombres ont une valeur absolue élevée, moins ils vont avoir de chances d'apparaître (le fameux principe qu'entre deux blocs consécutifs, les nombres varient peu). Ça ne vous rappelle rien cette remarque ? Non ? Alors reportez-vous à la page 44 sur le code de Golomb-Rice. C'était en effet précisément l'idée sous-jacente de ce code, et c'est donc avec une de ses variantes que l'on va coder les différences. On va pour cela utiliser la table suivante :

ligne	code unaire	nombres
0	0	0
1	10	-1 1
2	110	-3 -2 2 3
3	1110	-7 -6 -5 -4 4 5 6 7
4	11110	-15 $-14\ldots-9$ -8 8 9$\ldots$14 15
5	111110	-31 -30 $-29\ldots-17$ -16 16 17$\ldots$29 30 31
6	1111110	-63 -62 $-61\ldots-33$ -32 32 33$\ldots$61 62 63
7	11111110	-127 $-126\ldots-65$ -64 64 65$\ldots$126 127
$\vdots$	$\vdots$	$\vdots$
14	111111111111110	-16383 $-16382\ldots-8193$ -8192 8192 8193$\ldots$16382 16383
15	1111111111111110	-32767 $-32766\ldots-16385$ -16384 16384 16385$\ldots$32766 32767
16	11111111111111110	32768

TAB. 22: Table permettant de coder les différences dans JPEG

Ainsi, le nombre d'index $(0,0)$ de la première DCT, à savoir 1197, se trouve sur la $11^{\text{ème}}$ ligne du tableau, et sur la $1197^{\text{ème}}$position, il sera donc codé 111111111110|10010101101, la première partie du code de Golomb correspondant au numéro de ligne (code unaire) et la deuxième partie représentant le numéro de colonne

(représentation binaire sur 11 bits, la première colonne étant le numéro 0). La deuxième DCT ayant une différence de -7 sera codée 1110|000, et la différence de la troisième étant 3, elle sera codée 110|11. Remarquez que la longueur de la deuxième partie du code correspond exactement au numéro de ligne du tableau. Ainsi un «0» sera encodé «0|».

Passons maintenant à la deuxième étape. Son intérêt est évident lorsque l'on regarde les DCT après quantification :

75	-2	-2	0	-1	0	0	0
-1	1	-1	0	0	0	0	0
2	0	-1	1	0	0	0	1
4	1	-1	0	0	0	0	0
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

On s'aperçoit que l'on retrouve des nombres identiques sur les lignes diagonales sud ouest–nord est. Donc, si on veut utiliser du RLE dans la troisième étape, on n'a pas intérêt à lire la matrice ligne par ligne ou colonne par colonne, mais plutôt en diagonale, comme le montre la figure 16. Il n'est pas très compliqué de réaliser

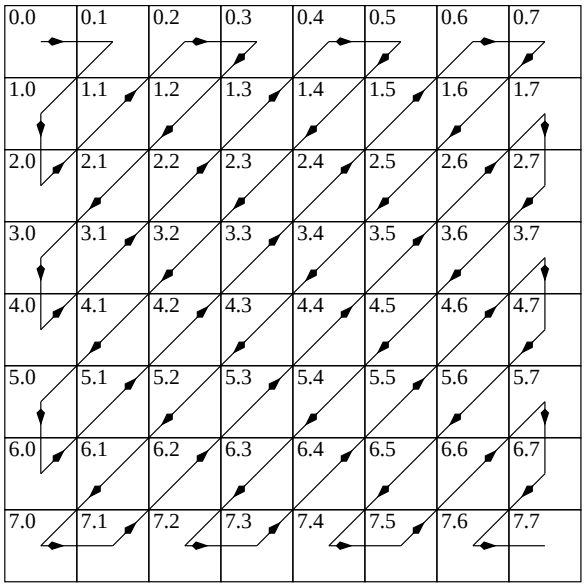


FIG. 16 – Le parcours de la DCT par l'algorithme JPEG.

un programme qui calcule ce parcours. Cependant, pour des raisons d'efficacité, on préfère utiliser une table représentant la séquence (il faut bien songer que les calculs seraient effectués très souvent et risqueraient donc de ralentir énormément l'algorithme de compression) :

```
typedef struct {
  int row;
  int col;
} zigzag;

zigzag ZigZag [N * N] = {
  {0, 0},
  {0, 1}, {1, 0},
  {2, 0}, {1, 1}, {0, 2},
  {0, 3}, {1, 2}, {2, 1}, {3, 0},
  {4, 0}, {3, 1}, {2, 2}, {1, 3}, {0, 4},
  {0, 5}, {1, 4}, {2, 3}, {3, 2}, {4, 1}, {5, 0},
  {6, 0}, {5, 1}, {4, 2}, {3, 3}, {2, 4}, {1, 5}, {0, 6},
  {0, 7}, {1, 6}, {2, 5}, {3, 4}, {4, 3}, {5, 2}, {6, 1}, {7, 0},
  {7, 1}, {6, 2}, {5, 3}, {4, 4}, {3, 5}, {2, 6}, {1, 7},
  {2, 7}, {3, 6}, {4, 5}, {5, 4}, {6, 3}, {7, 2},
  {7, 3}, {6, 4}, {5, 5}, {4, 6}, {3, 7},
  {4, 7}, {5, 6}, {6, 5}, {7, 4},
  {7, 5}, {6, 6}, {5, 7},
  {6, 7}, {7, 6},
  {7, 7}};
```

Le parcours et la quantification de la DCT peut alors s'effectuer grâce au petit bout de code suivant :

```
for (i=0, N2 = N * N; i< N * N; i++)
  {
    row = ZigZag[i].row;
    col = ZigZag[i].col;
    DCT[row][col] = rint (DCT[row][col] / Quantum[row][col]);
  }
```

Il reste maintenant à encoder les éléments autres que celui en haut à gauche de la DCT. Pour cela, JPEG utilise le parcours en zigzag et les 4 étapes suivantes : pour chaque élément non nul X de la DCT,

1. JPEG calcule le nombre Z d'éléments à zéro avant X (en suivant le parcours en zigzag, bien entendu) ;
2. il calcule la colonne C et la ligne L correspondant au nombre X dans la table 22 ;
3. le couple (Z, L) lui donne un nombre Y dans la table de Huffman 23 ;
4. enfin, on concatène à Y le nombre C codé sur L bits et on envoie le tout sur le flot de sortie.

(Z, L)	code	(Z, L)	code	...	(Z, L)	code
(0,0)(EOB)	1010			...	(15,0)(ZRL)	11111111001
(0,1)	00	(1,1)	1100	...	(15,1)	1111111111110101
(0,2)	01	(1,2)	11011	...	(15,2)	1111111111110110
(0,3)	100	(1,3)	1111001	...	(15,3)	1111111111110111
(0,4)	1011	(1,4)	111110110	...	(15,4)	1111111111111000
(0,5)	11010	(1,5)	11111110110	...	(15,5)	1111111111111001
⋮	⋮	⋮	⋮	⋮	⋮	⋮

TAB. 23: Table associant Y au couple (Z, L)

EOB est le code émis pour indiquer la fin d'un bloc et ZRL celui émis pour indiquer 15 zéros consécutifs lorsque le nombre de zéros consécutifs excède 15. Pour en finir avec ce codage, et avec le cours de compression en général, examinons juste un petit exemple des 4 étapes mentionnées ci-dessus. Supposons que nous voulions encoder le nombre $X = -6$, qui était précédé de 15 zéros (suivant le sens de parcours en zigzag). Alors $Z = 15$ et, conformément à la table 22, $L = 3$ et $C = 1$. Par conséquent, d'après la table 23, on doit envoyer sur le flot de sortie la séquence 1111111111110111|001. De même, si l'on a $X = 2$ précédé d'aucun zéro, alors $Z = 0$, $L = 2$ et $C = 2$. Par conséquent, on émettra sur le flot de sortie la séquence 01|10.

Index