MASTER EN SCIENCES INFORMATIQUES

Études sur la compression de données

Hody, Serge

Award date:
1987

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix

Institut d'Informatique

Année académique 1986-1987

Etudes sur la compression de données

Serge Hody

Mémoire présenté en vue de l'obtention

du grade de Licencié et Maître en informatique

Trois méthodes, dont une originale, de compression globale de données, leurs bases, une implémentation et leurs variations sont proposées. Les résultats de ces méthodes, isolées ou combinées, sont analysés et comparés, et un algorithme combinant deux approches est proposé.

Three global oriented Data-Compression methods, including an original one, their basis, implementation and variations are proposed. The results of these methods, isolated or combined, are analyzed and compared, and an algorithm combining two approaches is proposed.

Remerciements

Je désirerais remercier ici les personnes qui m'ont aidé dans la réalisation de ce mémoire, et le stage qui l'a précédé :

Monsieur Jean Ramaekers, directeur de ce mémoire, pour sa disponibilité et ses conseils éclairés.

Ma famille, et spécialement mes parents, pour leur soutien et patience inlassables.

Messieurs Harald Luedtke et Olivier Staes, pour la lecture attentive du manuscrit et leurs remarques parfois féroces.

Monsieur André Yakovleff, pour sa collaboration enthousiaste matérialisée dans l'annexe 1 de ce mémoire.

Mesdemoiselles Valérie Mancini, Françoise Miane, Anne Sauvage et Sylvie Mary pour leur aide et leur soutien permanents.

Monsieur Dieter Hein, pour sa compréhension et les facilités accordées lors de la rédaction de ce mémoire.

Les membres du personnel de Siemens Software Namur, et notamment Madame Guillaume, et Messieurs Radelet, Brison et de Cocqueau.

Les membres du personnel de Siemens A.G. München, et notamment Mesdames Reich, Pogoda, Polin, Rung, Weber, Lammers, Christman, Galneder, Markauser, Legardeur, Tuchscheerer, et Messieurs Reith, Kost, Hein, Herlet, Neubacher, Heger, Boëhm, Layer, Stöker, Albrecht, Kellerer, Dulich, Potthast, Torno, Legardeur, Flament, Marquis, Maximillien, Guyonnet, Schmitz, Knecht, Picarellio, Kinderstut.

Je désirerais aussi remercier ceux qui m'ont aidé dans les cinq années précédentes :

Les membres du corps professoral de l'Institut d'Informatique et de la Faculté des Sciences Economiques et Sociales, et notamment Messieurs Drabs, Guillaume et Hainaut, qui m'a un jour ouvert les yeux.

Le personnel du secrétariat de l'Institut d'Informatique, et notamment Madame Decoux, pour leur infinie gentillesse.

Thierry, Anne, Nathalie, Calou, Pierre, Philippe, Bart, Uli, Sylvia, Brigitte, Evelyne et Monica.

Mademoiselle Fabienne Bister.

Je remercie enfin tous ceux qui m'ont aidé, et que je n'ai pas remercié ici par oubli ou manque de place. Il y a tellement de gens formidables.

Ce mémoire est dédié à Monsieur Jean-Marc Ménestret, décédé le premier octobre 1983.

Table_des_matières

0. Introduction

Une caractéristique dominante de la science informatique a toujours été la recherche de la performance. La performance, qui signifiait hier place mémoire utilisée et temps de réponse minimaux, s'est aujourd'hui transformée, devant l'apparition de systèmes disposant de mémoires virtuelles toujours plus grandes (la dernière version du système SIEMENS BS2000 est dotée d'un espace adressable par 31 bits), en une notion plus qualitative d'appréciation du service fourni dans laquelle est inclus le temps de réponse.

Ceci ne signifie pourtant pas que la notion d'espace utilisé ne joue plus aucun rôle. En effet, si les coûts, en termes de CPU et de mémoire principale, ne cessent de baisser, il n'en va pas de même du prix des mémoires de masse, ni surtout des télécommunications. Additionnellement, le volume d'information a un rapport immédiat avec son temps de transfert et donc le temps de réponse, tandis que l'optimisation du volume occupé par une base de données a une influence immédiate sur ses performances, par la minimisation des mouvements entre les mémoires principales et secondaires [CORMACK 1985].

Cette préoccupation se réflète d'ailleurs dans l'orientation des publications scientifiques concernant ce sujet, qui semble indiquer que la compression de données est devenu un sujet de recherche à part entière. Ainsi, les études s'y rapportant, autrefois essentiellement simples et pratiques ([SNYDERMAN 1970], [RUTH 1972]), laissent aujourd'hui apparaître un soucis de recherche de bases théoriques ([STORER 1983]), tandis que voient le jour des méthodes complexes faisant appel aux ressources du

parallélisme ([GONZALEZ 1985]).

Face aux performances parfois étonnantes observées dans la littérature et à l'absence fréquente de mécanismes d'optimisation dans de nombreux systèmes (SIEMENS BS2000, DEC TOPS-20), plusieurs questions peuvent être posées :

- ces mécanismes d'optimisation (compression de données) peuvent-ils être appliqués de manière générale, ou existe-t-il une relation étroite entre les gains réalisés et la nature des données ?

- quelle est la complexité de ces mécanismes ?

- une compréhension des bases logiques de la compression de données ne permettrait-elle pas une adaptation ("customisation") des procédures aux données particulières du sytème ?

- n'ont-ils pas d'autres inconvénients que les simples coûts de compression et de décompression ?

Dans ce mémoire, nous allons donc tenter de trouver une réponse à ces questions et pour cela proposer une étude de trois mécanismes de compression se basant sur des caractéristiques différentes des données, en décrivant leurs bases logiques, l'implémentation des procédures de compression et de décompression, ainsi que leurs variantes et leur emplacement logique dans le système. Ces méthodes, essentiellement destinées à une compression globale des données à des fins d'archivage ou de transfert, peuvent être adaptées à des ensembles mouvants, telles que les bases de données.

Pour chacune de ces méthodes, nous proposerons des algorithmes de prévision du volume de compression réalisable.

Dans une première partie (chapitre 1 et 2), nous définirons une terminologie et une base logique de la compression de données (chapitre 1) et expliquerons la notion de prévision du volume de compression et son utilité (chapitre 2).

Dans la seconde partie (chapitre 3 à 6), nous proposerons l'analyse de trois méthodes de codage, leurs bases logiques, une implémentation possible de ces méthodes, et quelques-unes de leurs variations. Tout d'abord, nous analyserons une méthode basée sur le contexte (chapitre 3) : la méthode Run-Length. Une méthode de codage basée sur les propriétés statistiques des données sera examinée dans les chapitres 4 et 5 : la méthode Huffman. Enfin, une approche originale basée sur la signification logique des données sera développée en chapitre 6.

Dans la troisième partie (chapitre 7 et 8), nous comparerons ces méthodes de manière quantitative et qualitative et proposerons un algorithme combiné permettant une amélioration à la fois de la compression réalisée et du temps (CPU et Entrées-Sorties) nécessaire à cette compression.

Nous terminerons (chapitre 9) par quelques considérations sur l'emplacement dans le système des routines de compression et de décompression.

1. La Compression de Données : Définitions et Principes de Base

Lorsque l'on observe la richesse de la littérature traitant du problème de la compression de données, il est surprenant de remarquer que peu de systèmes utilisent ces mécanismes de manière intensive.

Un tel besoin existe pourtant : le coût des périphériques de stockage permanent de l'information, et plus spécialement les périphériques à accès directs (disques), semblent devenir aujourd'hui le point noir des budgets des centres de traitement de l'information.

Ainsi que l'a souligné Dennis G. Severance dans un excellent article [SEVERANCE 1983], "Trois faits aident à expliquer ce phénomène :

(1) Les concepteurs sous-estiment habituellement la quantité de compression réalisable pour une base de données déterminée, et les implications de cette compression ne sont pas pleinement appréciées.

(2) La Compression de données ajoute un niveau de complexité à la conception, à l'implémentation, et au fonctionnement d'un système d'information, et les concepteurs sont réticents à accepter une complexité additionnelle sans des bénéfices clairs et substantiels.

(3) L'essentiel de la littérature publiée traite de techniques de compression particulières, les entourant d'une mystique mathématique. Les praticiens évitent de manière bien compréhensible des domaines dans lesquels ils ne sont pas à l'aise."

Avant de poursuivre, nous allons définir certains concepts et termes se rapportant à la compression de données. Nous utiliserons pour l'essentiel la terminologie de Severance.

1.1 La Terminologie

Le Codage de Données est "un Processus établissant une relation entre une collection d'unités de codage (un ou plusieurs symboles formant une représentation de données) et une collection de valeurs de code (un ou plusieurs symboles formant une autre représentation de données). La relation existant entre unités de codage et leurs valeurs de code correspondantes est appelée un code. Si la relation est de type 1-1, alors une relation inverse existe et le processus inverse est appelé décodage.

La Compaction de Données est une forme de codage réduisant° la taille (en terme de bits) des données.

La Compression de Données est un processus de compaction réversible, et donc sans perte°° d'information.. [SEVERANCE 1983]

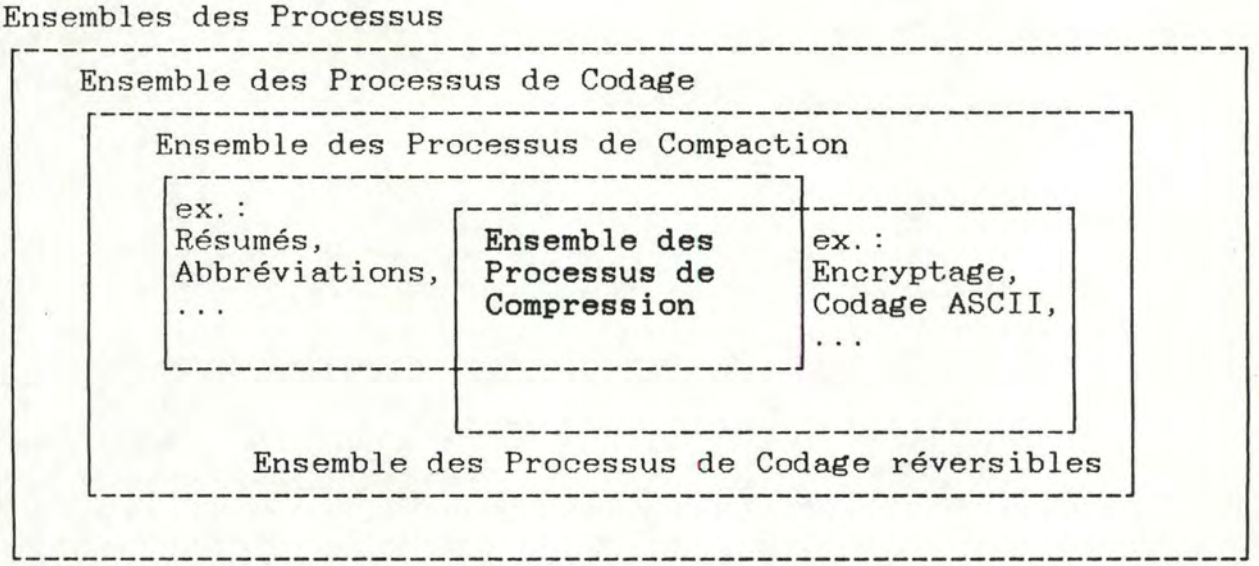
$$\text{Ratio de Compression} = \frac{\text{Longueur originale des données}}{\text{Longueur des données comprimées}}$$

$$\text{Figure Of Merit} = \frac{1}{\text{Ratio de Compression}}$$

° Une expansion peut se faire dans le cas de données peu compressibles, ceci étant dû aux informations (tables,...) nécessaires au processus de compaction lui-même.

°° Les résumés, abbréviations, etc. peuvent être non-réversibles de par la perte d'information.

Nous pouvons représenter ces concepts graphiquement :



(Figure 1.1)

Cette terminologie n'est certes pas généralisée dans la littérature; les termes "Compression" et "Compaction", notamment, sont souvent utilisés comme des synonymes. Nous utiliserons par la suite le terme "Compression" au sens défini ici, à savoir comme synonyme de "Compaction réversible".

1.2 Les Bases

Sans s'étendre trop sur des considérations théoriques (le lecteur intéressé les trouvera dans [SHANNON 1948], [STORER 1983]), il est possible d'établir des liens directs entre la théorie de l'information et les méthodes pratiques étudiées par la suite.

A partir de la théorie de Shannon, on peut aisément déterminer que le nombre optimal B de bits nécessaires à l'encodage d'un symbole S est donné par la formule

$$B = - \log_2 p(S) \qquad \text{(Expression 1.1)}$$

avec p(S) = probabilité d'occurence du symbole S

Il est à noter que ce nombre B est habituellement impossible à atteindre en représentation digitale, de par la partie fractionnelle de B, mais nous étudierons dans le chapitre 4 une méthode permettant d'atteindre un code qui est aussi proche que possible de cet optimum.

Nous montrerons de plus que d'autres méthodes de compression ont une relation avec la théorie de l'information, de par leur manière de considérer les données non plus selon une vue statistique globale, mais selon une séquence de caractères reliés logiquement. Cette relation, parfois évidente (méthode Run-Length), parfois plus complexe (méthode LOVER), sera expliquée plus en détail dans les chapitres concernés.

Intuitivement, l'expression 1.1 signifie que plus la probabilité d'un symbole est élevée ($0 \leq pr \leq 1$), plus la longueur du code doit être faible, avec comme cas extrêmes :

pr = 0 : le code a une longueur infinie

pr = 1 : le code a une longueur nulle, ce qui signifie que le symbole ne transporte aucune information .

Par conséquent, si nous assignons des codes courts aux symboles les plus fréquents, et des codes plus longs aux autres caractères, il en résultera un gain dans la taille des données.

Plus formellement, nous pouvons dire que :

$\forall F \in$ { Processus de Codage },

avec O_F = longueur des informations de service ("Overhead") introduites par F

$\forall D_I$, quantité d'information donnée, représentée dans un code C_I

avec $L(D_I) = L_I$ = Longueur (en bits) de D_I

avec $I(D_I) = I_I$ = Information contenue dans D_I

si $D_O = F(D_I)$, représentée dans un code C_O

avec $L(D_O) = L_O$ et $I(D_O) = I_O$

On peut dire que :

1. F est un processus de codage réversible si et seulement si il existe $F^{-1} \in$ { Processus de codage } et

$F^{-1}(D_O) = D_I$ (=> $I_O \geq I_I$)

2. F est un processus de Compaction si et seulement si

$$L_O - O_F \leq L_I$$

3. F est un processus de Compression si et seulement si F est un processus de codage réversible & F est un processus de Compaction

Il est intéressant de noter que les processus de compaction n'ont habituellement pas la propriété :

$$F(F^{-1}(D_I)) = F^{-1}(F(D_I)) = D_I$$

(Expression 1.2)

Ceci parce que F^{-1} a généralement besoin d'une information de service générée par F, telle des tables de code, etc.

2. L'Analyse et la Prédiction

De nombreuses applications dans lesquelles la compression de données pourrait avoir un rôle important à jouer, ont une caractéristique commune : l'automatisation. Des fonctionnalités telles que l'archivage des fichiers, le transfert de données, ou les mémoires de masse hiérarchiques, se font sans l'intervention de l'opérateur, et donc sans paramétrage précis des caractéristiques des données, qui influencent grandement le pourcentage de compression réalisable, ainsi que nous le verrons au chapitre 7.

Le choix de l'algorithme de compression, d'efficacité assez variable, est donc un point crucial duquel va dépendre toute l'efficacité du processus.

De plus, si le but principal est la réduction de taille des données, les facteurs temps de compression (CPU et Entrées-Sorties) et intégrité des données ne doivent pas être négligés : à ratios de compression comparables, la méthode Run-Length devra être préférée à la méthode Huffman, plus coûteuse et plus fragile.

Nous allons donc développer, pour chacune des méthodes (à l'exception de la méthode adaptative Huffman, par nature, et par fonction, imprévisible), des algorithmes nous permettant de prédire en une seule lecture du fichier le taux de compression prévisible, et de là nous permettant de sélectionner l'algorithme de compression adéquat. Ces algorithmes seront explicités dans les chapitres se rapportant à chacune des méthodes.

3. Un Codage sur Base du Contexte : la Méthode Run-Length

La méthode de codage Run-Length n'est certainement pas la seule méthode de compression de données se basant sur le contexte dans lequel apparaît un caractère; des techniques telles LOVER (voir chapitre 6), ou encore des algorithmes de compression de programmes sources [KATAJAINEN 1986], peuvent être considérées elles aussi comme basées sur le contexte.

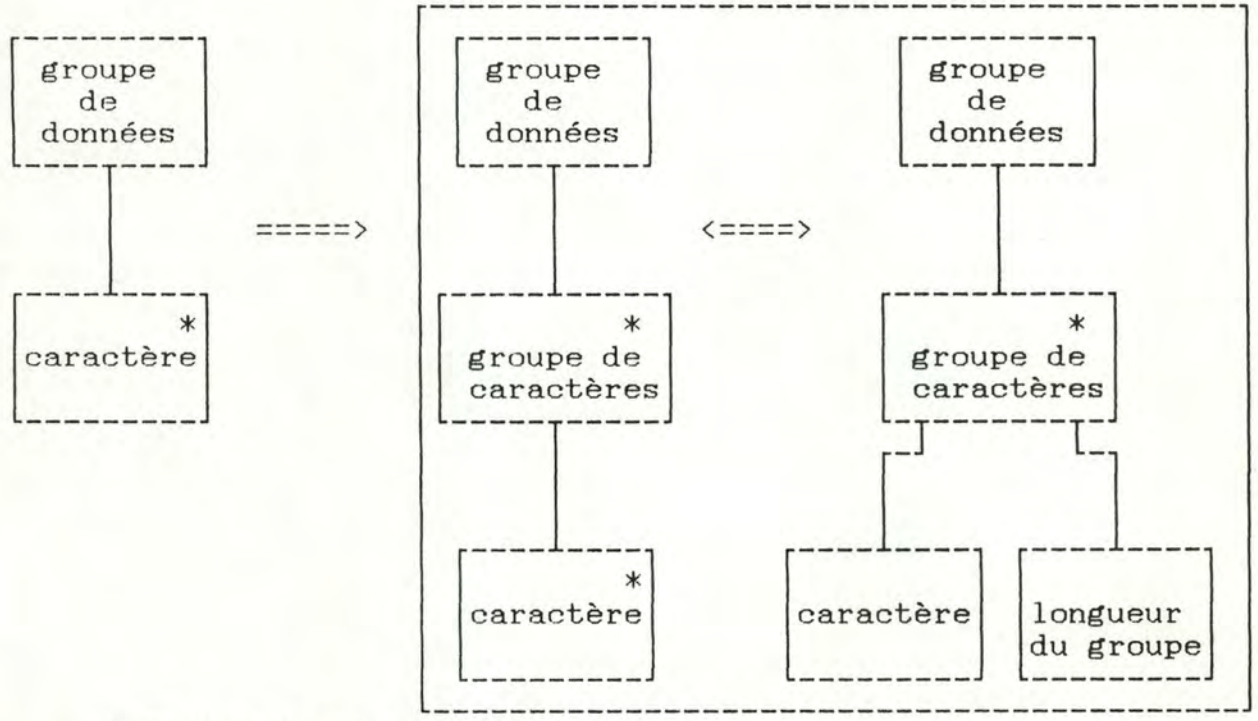
Mais plus que basées sur le contexte, elles sont orientées vers des groupes logiques, structurés, d'information, ou plus encore la syntaxe des données; elles sont en général plus complexes, et plus proches de techniques de compilation telles celles décrites dans [AHO 1977].

Ainsi que l'écrit Held dans [HELD 1983], "La méthode de codage Run-length est une méthode de compression de données réduisant physiquement toute séquence répétitive de caractère lorsque la longueur de cette séquence atteint un niveau prédéfini.

Une variation connue, ou plus exactement un sous-ensemble de cette technique, est la suppression du caractères NULL (ASCII 0), qui est utilisée dans le protocole de transmission IBM 3780 BISYNC. Cette technique simple a peu d'avantages, et donne généralement des ratios de compression moins satisfaisants. Le lecteur pourra en trouver une description détaillée dans [HELD 1983].

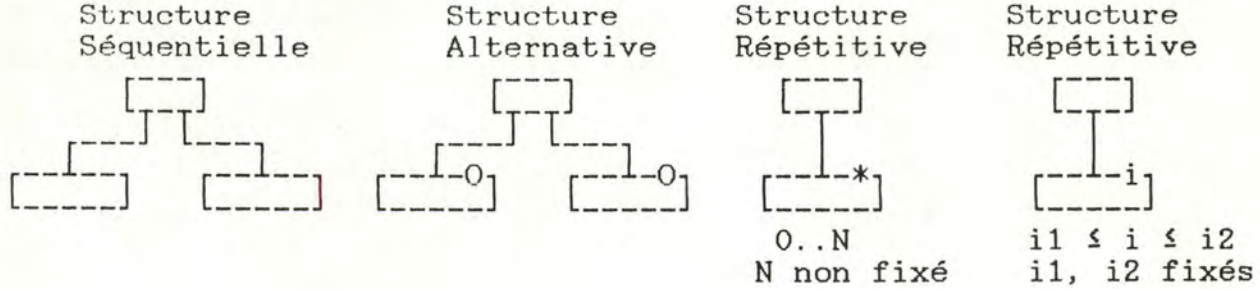
3.1 Les Bases théoriques

L'idée de base de cette méthode est qu'un ensemble de données ne devrait pas être regardé comme une succession de caractères, mais bien comme une séquence de groupes d'un même caractère ; en utilisant la méthode Jackson (voir [JACKSON 1975]), nous pouvons représenter une telle structure graphiquement de la manière suivante :



(Figure 3.1)

Signification des Structures graphiques



Ceci correspond à un principe très utilisé en description d'ensembles : l'énumération complète de l'ensemble est remplacée par le couple { nombre d'éléments de l'ensemble, propriété(s) décrivant complètement les membres de l'ensemble } ; dans ce cas, la seule propriété de ces éléments est leur code.

Par référence à la théorie de l'information, cette vision de l'ensemble de données nous permet de conclure qu'un groupe de caractères est entièrement décrit par le caractère lui-même et la longueur du groupe, c'est-à-dire que le nombre de bits nécessaire à décrire une chaîne de caractères équivalents est égal à la somme du nombre de bits nécessaire pour décrire le caractère, et du nombre de bits nécessaire pour décrire la longueur, ou, pour reprendre l'expression de Shannon :

```
I(Groupe de caractères)
                = I(Caractère) + I(longueur du groupe)
avec I(X) = information (en bits) contenue dans X
```

(Expression 3.1)

Cette valeur est aisément calculable : si nous considérons que chaque type de groupe de caractères (séquences de 'a','b',...) a la même probabilité d'occurence, et donc transporte la même quantité d'information, chaque caractère devrait être codé sur 7 ou 8 bits, dépendant du code utilisé : ASCII 7 ou 8 bits, EBCDIC 8 bits, etc. (Nous ne considérerons plus par la suite que des codes 8 bits). De plus, si nous considérons qu'aucun groupe n'a une longueur supérieure à 256 caractères, nous pouvons coder cette longueur sur un octet, nous

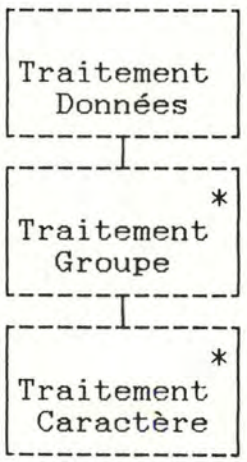
donnant une longueur totale de 2 octets (16 bits) par groupe de caractères.

Nous examinerons dans ce chapitre comment éviter le codage de groupes ne comportant qu'un seul caractère, en cassant le modèle "pur", et nous étudierons dans le chapitre 8 une méthode nous permettant d'optimiser le codage, en tenant compte des différences fréquentielles entre la nature de chaque groupe d'une part, et entre leur longueur d'autre part.

3.2 L'Implémentation

A. Le Codage

Suivant les théories de Jackson et Warnier (voir [JACKSON 1975], [BERGLAND 1981], [HIGGINS 1977], [HIGGINS 1979]) que le traitement devrait être guidé par la structure des données, l'algorithme est immédiat :




```
Program Process_File.
Begin
     Open File_In, File-Out;
     End_File:="F";
     If not EOF(File_in) then Read(File_in,Car)
                         else End_File:="T";
     While (End_File<>"T") do Process_Group;
     Close File_In, File_Out;
end.

Procedure Process_group;
Begin
     Oldcar:=Car;
     Length_Run:=0;
     While ((Car=Oldcar) & (Length_Run<255) & (End_File<>"T"))
     do Process_char;
     Write(File_Out,Oldcar,Length_Run);
end;

Procedure Process_Char;
Begin
     If not EOF(File_in) then Read(File_in,Car)
                         Else End_File:="T";
     If (Car=Oldcar) then Length_Run:=Length_Run + 1;
end;
```

(Algorithme 3.1)

La vue des données en tant que succession de groupes de caractères n'est pourtant pas optimale du point de vue de la compression : tous les groupes, même ceux ne comportant qu'un seul caractère, sont codés sur deux octets; il en résulte usuellement une expansion des données.

Une solution immédiate est de ne comprimer que les groupes d'une longueur supérieure à deux caractères. Mais la nécessité de distinguer entre les groupes comprimés et ceux qui ne le sont pas nécessite l'utilisation d'un caractère spécial, et nous amène à ne compacter que les groupes de plus de 3 caractères, ou les groupes constitués du caractère spécial utilisé.

```
Program Process_File.
Begin
     Open File_In, File-Out;
     End_File:="F";
     Write(File_out,Special_char);
       /* le décodeur doit connaître le caractère spécial utilisé
     If not EOF(File_in) then Read(File_in,Car)
                         else End_File:="T";
     While (End_File<>"T") do Process_Goup;
     Close File_In, File_Out;
end.

Procedure Process_group;
Begin
     Oldcar:=Car;
     Length_Run:=0;
     Write(File_out,Special_Char);
     While ((Car=Oldcar) & (Length_Run<255) & (End_File<>"T"))
     do Process_char;
     if ((Oldcar=Special_char) or (Length_Run >= 4)
     then Write(File_out,Special_Char,Oldcar,Length_Run)
     else for i:=1 to Length_Run
          do Write(File_Out,Oldcar);
end;

Procedure Process_Char;
Begin
     If not EOF(File_in) then Read(File_in,Car)
                         Else End_File:="T";
     If (Car=Oldcar) then Length_Run:=Length_Run + 1;
end;
```

(Algorithme 3.2)

Cette simple modification de la structure de base nous donne un codage optimal, dans l'hypothèse d'un codage à longueur déterminée. Face à la petite quantité d'overhead introduite (un seul caractère au début du fichier), on pourrait s'interroger sur la possibilité de l'application répétée de cette méthode. Malheureusement, cette méthode détruit la propriété sur laquelle elle se base, à savoir la succession de caractères semblables, et donc, à l'exception de cas singuliers, ne peut être appliquée plusieurs fois avec succès.

La question du choix du caractère spécial, souvent effectué de manière arbitraire, sera abordée de manière plus générale dans le paragraphe suivant.

La combinaison de cette méthode avec le codage statistique et LOVER, sera étudiée dans les chapitres 7 et 8.

B.Le_Décodage

Dans le modèle initial, l'algorithme est immédiat :

```
Program Decompress_File.

Begin
     Open File_in, File_out;
     End_File:="F";
     Read_a_Group(Car,Length,End_File);
     While (End_file<>"T")
     do Process_Group(Car,Length);
        Read_a_Group(Car,Length,End_File)
     end;
     close File_in, File_out;
end.

Procedure Read_a_Group;

Begin
     If not EOF(File_in)
     then read(File_in,Car,Length)
     else End_File:="T";
end;

Procedure Process_Group;

Begin
     For i:=1 to Length do write(File_out,car);
end;
```

(Algorithme 3.3)

L'optimisation du codage n'entraîne que peu de modifications, localisées dans la procedure Read_a_Group :

```
Procedure Read_a_Group;

Begin
     If not EOF(File_in)
     then begin
               read(File_in,Car);
               if (Car=Special_char)
               then Begin
                         read(File_in,car);
                         read(File_in,Length);
                    end
               else length:=1
          end
     else End_File:="T";
end;
```

(Algorithme 3.4)

3.3 La Prédiction

Cette algorithme reflètera lui-aussi la structure des données ; la solution est triviale dans la structure de base :

- Comptage du nombre de groupes dans les données en entrée ;
- Données en sortie = nombre de groupes X (2 caractères/ groupe)

Le lecteur remarquera que les modifications dans l'algorithme 3.1 sont mineures.

Dans la structure optimisée, les changements sont là aussi peu importants :

```
Program Process_File.
Begin
     Open File_In;
     End_File:="F";
     If not EOF(File_in) then begin
                                   Read(File_in,Car);
                                   Size_Output:=1
                              end
                         else begin
                                   End_File:="T";
                                   Size_Output:=0
                              end;
     While (End_File<>"T") do Process_Goup;
     write('Size Output = ',Size_Output);
     Close File_In;
end.

Procedure Process_group;
Begin
     Oldcar:=Car;
     Length_Run:=0;
     While ((Car=Oldcar) & (Length_Run<255) & (End_File<>"T"))
     do Process_char;
     if (Length_Run >= 4)
     then Size_Output:=Size_Output + 3
     else Size_Output:=Size_Output + Length_Run;
end;

Procedure Process_Char;
Begin
     If not EOF(File_in) then Read(File_in,Car)
                         Else End_File:="T";
     If (Car=Oldcar) then Length_Run:=Length_Run + 1;
end;
```

(Algorithme 3.5)

Le lecteur pourra noter dans l'algorithme 3.5 une approximation apparente : tous les groupes (inclus les groupes de caractère spécial) d'une longueur inférieure à quatre ne sont pas considérés comme codés. Cela est pourtant correct, si on choisit le caractère spécial de telle manière qu'il apparaît le moins fréquemment possible dans des groupes de longueur inférieure à quatre.

Le choix de ce caractère spécial pourra donc s'effectuer de telle manière qu'il apparaisse le moins souvent possible (idéalement 0 fois) dans des groupes de longueur $L < 4$.

Ceci est réalisé assez facilement, par l'utilisation d'une table T de 256 éléments, définie comme suit :

$\forall\ t[i] \in T,\ 0 \leq i \leq 255$:

t[i] = nombre de groupes du caractère i, de longueur $L < 4$

(Algorithme 3.6)

Le choix du caractère se bornera donc à la recherche de l'emplacement de l'élément le plus petit de la table.

4. Un Codage statistique : l'Algorithme Huffman

L'Algorithme Huffman (AH) est certainement l'algorithme se référant à la compression de données le plus étudié depuis son apparition en 1952.

Même si, paradoxalement, l'article original [HUFFMAN 1952] peut être difficile à trouver, le lecteur en trouvera l'étude complète, les bases, les commentaires et des variations dans [SHANNON 1948], [RUTH 1972], [WELLS 1972], [GALLAGER 1978], [CORBIN 1981], [PECHURA 1982], [SEVERANCE 1983], [CORMACK 1984], [MCINTYRE 1985], [HELD 1983], [SEDGEWICK 1983], [AMSTERDAM 1986], [TANENBAUM 1981].

Nous allons tout d'abord proposer une définition algorithmique simple des codes Huffman, et prouver l'optimalité du code généré par cet algorithme, et les limites de cette optimalité. Nous établirons ensuite deux propriétés qui nous permettront d'en donner une définition structurelle.

Dans une seconde partie, nous proposerons une implémentation des algorithmes de codage et de décodage.

Enfin, nous proposerons une méthode permettant de prédire le ratio de compression, et nous terminerons avec deux variations de la méthode originale.

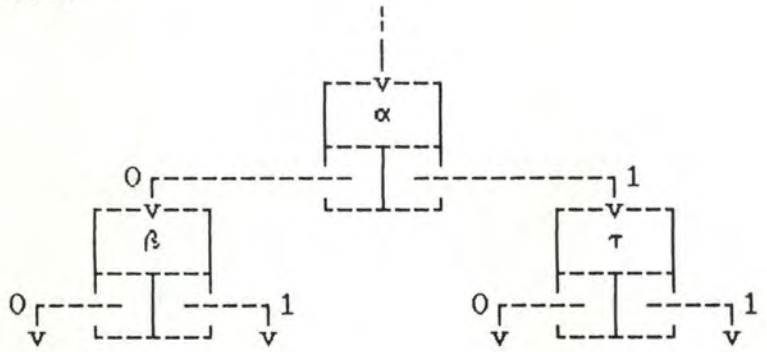
4.1 Les Bases théoriques

Dans un système digital, l'alphabet est représenté sous forme d'un code binaire, de longueur L fixe pour des raisons de facilité de manipulation. Un tel code peut être regardé comme un arbre binaire à L niveaux, dont les feuilles sont les caractères du code, et dont le chemin de la racine vers la feuille forme le code du caractère. Il est à noter qu'un tel code possède une propriété de préfixe, c'est-à-dire qu'aucun caractère n'a un code qui soit le préfixe du code d'un autre caractère. L'idée de base de l'AH est la création d'un arbre dont les feuilles se trouvent à des niveaux différents de l'arbre, tout en conservant cette propriété de préfixe.

Ainsi que nous l'avons noté au chapitre 1, l'entropie d'un symbole S, définie comme la quantité d'information transportée par ce symbole, et exprimée par (expression 1.1), est habituellement impossible à atteindre, ceci étant dû à la partie fractionnelle de cette expression.

L'AH nous permet d'approcher cette valeur de la manière la plus proche possible dans un système de représentation digitale.

Définissons comme Siblings des éléments ayant le même parent immédiat dans l'arbre ; une représentation graphique de cette relation pourrait être :



(Figure 4.1)

Dans cette structure, ß et τ sont siblings, α est le parent de ß et τ, qui sont ses enfants.

Algorithme_Huffman

" 1) Soit L, une liste des probabilités des symboles sources correspondant aux feuilles de l'arbre de codage.

2) Prendre les deux éléments de L ayant la probabilité la plus faible, les faire Siblings, générer un noeud intermédiaire en tant que parent de ces siblings, marquer le lien du parent vers un enfant avec la valeur 0, et l'autre lien avec la valeur 1.

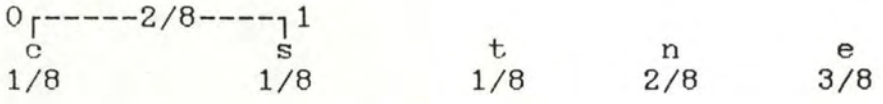
3) Remplacer les deux éléments choisis par le parent, avec une probabilité égale à la somme des probabilités des deux éléments. Si la liste L ne contient qu'un élément, arrêter, sinon retourner au point 2)." [GALLAGER 1978]

(Algorithme 4.1)

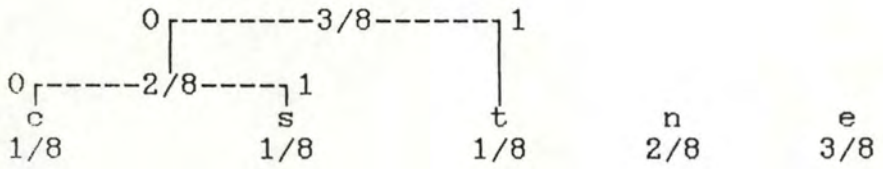
explicitons cet algorithme sur le mot "sentence".

lettres	c	s	t	n	e
fréquences	1/8	1/8	1/8	2/8	3/8

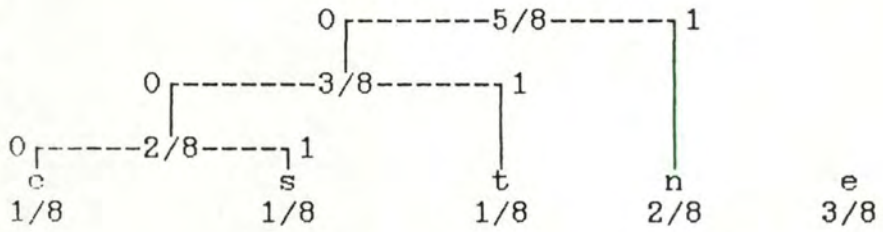
étape_1



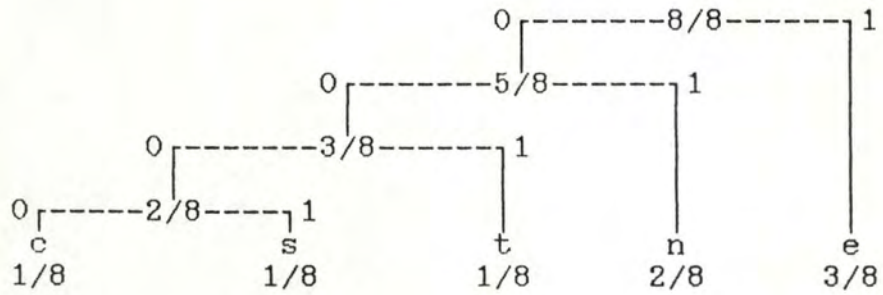
étape_2



étape_3



dernière_étape



Le code optimal est donc

e = 1 n = 01 t = 001 s = 0001 c = 0000

Preuve de l'optimalité du code généré

Il nous reste à montrer que le code généré est bien optimal, c'est-à-dire qu'il n'existe pas d'autre code permettant de représenter la même quantité d'information en un plus petit nombre de bits. Cette preuve, donnée par Huffman dans son article original, a été récemment reprise par Amsterdam dans [AMSTERDAM 1986].

Définissons tout d'abord le WPL (Weighted Path Length) d'un arbre de la manière suivante :

$$WPL = \sum_{i=1}^{k} (\text{nombre d'occurences du caractère Ci} * \text{Longueur du chemin de la racine au caractère Ci})$$

avec k = nombre de caractères de l'arbre

(Expression 4.1)

Il est clair que, pour un ensemble de données D, la représentation de D la plus petite sera celle codée suivant un arbre de code T de WPL minimal.

Nous allons montrer que, pour une distribution statistique donnée, l'AH produit un arbre de code de WPL minimal. Pour cela, faisons tout d'abord trois observations :

1) observation de généralité : on peut toujours construire un arbre de code en combinant répétitivement deux sous-arbres en un seul plus important. L'aspect important de l'AH est qu'il choisit toujours les deux sous-arbres de fréquences minimales.

2) observation d'abaissement : accroître la longueur du chemin jusqu'à un sous-arbre de 1 (= abaisser le sous-arbre) revient à additionner la fréquence du sous-arbre au WPL de l'arbre global.

3) observation d'échange : si le sous-arbre ST1 a une fréquence inférieure au sous-arbre ST2, et si le chemin de la racine à ST1 est plus court de n chemins élémentaires que celui de la racine à ST2, permuter ST1 et ST2 diminuera le WPL de l'arbre global; ceci est une conséquence directe de 2), et correspond à additionner n fois la fréquence de ST1, et à soustraire n fois la fréquence de ST2.

Démonstration

Considérons un algorithme créant un arbre de code en sélectionnant les éléments à faire Siblings de manière différente par rapport à l'étape 2 de l'algorithme 4.1 ; supposons que cet algorithme choisisse non pas les deux éléments F1 et F2 de fréquence la plus faible, mais l'élément F1 de fréquence minimale, et un élément F de fréquence supérieure.

Trois possibilités se présentent :

1) F2 se trouve plus bas dans l'arbre final que F1, le plus petit ; de par l'observation d'échange, on peut permuter F2 et F1, afin d'obtenir un arbre de code de WPL inférieur.

2) F2 se trouve plus haut dans l'arbre que F1, et donc que le Sibling de F1, qui est F, de fréquence plus grande que celle de F2 ; de par cette même observation, on peut permuter F2 et F, et donc obtenir un arbre de code de WPL inférieur.

3) F2 se trouve au même niveau que F1 et F ; on peut donc permuter F2 et F sans modification du WPL, et obtenir l'arbre qui aurait été généré par l'AH.

Il est clair que toute altération du type 1) ou 2) produit un code qui n'est pas optimal ; une altération de type 3) produit quant à elle un arbre qui est optimal, mais pas meilleur que celui qui aurait été généré par l'AH.

Considérons que l'algorithme choisisse les deux éléments de manière entièrement différente par rapport à la méthode de choix de l'algorithme Huffman. La démonstration se fait en deux étapes : tout d'abord, on montre qu'un algorithme choisissant l'élément de fréquence minimale, et un des deux autres éléments, produit un arbre de WPL meilleur, de la même manière que celle utilisée dans la première partie de la démonstration, ce qui nous ramène au présupposé de cette première partie. C.Q.F.D.

Nous avons donc prouvé que l'AH produit un code optimal. Il faut cependant remarquer que ce n'est pas toujours l'algorithme le plus efficient pour comprimer un ensemble quelconque de données. En effet, son ratio de compression atteint dans le meilleur des cas 8:1, cas d'un code dans lequel tous les caractères originaux sont codés sur un bit. En comparaison, un codage de type Run-Length peut atteindre un ratio de 256:3, cas d'un ensemble de données constitué de suites de 256 caractères. Ainsi, une image graphique, succession de chaînes de 0 et de 1, sera comprimée plus efficacement par un algorithme de type Run-Length opérant au niveau du bit, tandis qu'un algorithme orienté vers la structure logique tel LOVER, les codes Huffman modifiés

(voir 4.5) ou l'algorithme Frankenstein-Lidzba-Verfahrens [STUCKY 1986] seront préférés pour la compression d'une base de données.

Deux propriétés des codes Huffman - Une définition structurelle

1) La propriété préfixe, immédiate conséquence de l'assimilation des symboles aux feuilles de l'arbre de code, assure qu'aucun code de symbole n'est le préfixe d'un autre code. Cela permet un décodage immédiat de la source codée.

2) La propriété Sibling, que nous allons définir de la manière suivante :

CT, arbre de code, a la propriété Sibling

<=> 1) Chaque noeud de l'arbre, racine exceptée, a un Sibling

2) Les noeud peuvent être listés par ordre croissant non-strictement de fréquence, de telle manière que chaque noeud est adjacent dans la liste à son Sibling, ce qui, formellement exprimé :

Propriété Sibling
Pour un alphabet de K éléments, la liste, racine exceptée, comporte (2K-2) noeuds, et
$\forall i,\ 1 \leq i \leq K-1$: Les éléments (2i) et (2i-1) sont Siblings

(Expression 4.2)

3) Une définition structurelle

Théorème

Soit C, un code binaire possédant le propriété préfixe, d'arbre de code CT :

C est un code Huffman <=> CT possède la propriété Sibling

(Expression 4.3)

Démonstration

1) CT possède la propriété Sibling => C est un code Huffman

=> Les éléments (1) et (2) sont Siblings et feuilles de CT ; en effet, s'ils étaient des noeuds, leurs enfants auraient des fréquences inférieures à celles des éléments (1) et (2), ce qui est impossible, les éléments étant classés par ordre de fréquence croissante.

=> Les éléments (1) et (2) sont deux symboles sources de fréquences minimales, et on peut donc les combiner suivant l'étape 2) de l'algorithme 4.1.

=> Si nous les supprimons de CT, CT garde la propriété Sibling, et les feuilles de l'arbre de code réduit correspondant à (CT/{Elém(1),Elém(2)}), correspondent à la liste L de l'algorithme 4.1 après l'étape 3)

=> A chaque itération, l'algorithme 4.1 choisit comme Siblings des éléments qui sont Siblings dans CT

=> Si nous marquons les chemins dans l'arbre de code généré par l'algorithme 4.1 suivant les chemins dans l'arbre de code CT, les deux codes sont équivalents.

2) CT est un arbre Huffman => CT a la Sibling Property

Soit LS une liste initialement vide ; exécutons l'algorithme 4.1, modifié de telle manière qu'à chaque itération, il ajoute à LS les deux noeuds (le terme "noeud" incluant ici les feuilles) sélectionnés, le noeud de plus faible fréquence avant l'autre noeud.

=> LS est construit d'une manière telle que chaque noeud de CT est adjacent dans LS à son Sibling ; de plus, de par le choix effectué à l'étape 2) de l'algorithme 4.1, et par la manière dont les paires de noeuds sont placées dans LS, LS est classée par ordre croissant de fréquence des éléments. C.Q.F.D.

4.2 L'Implémentation

A. La procédure de compression

L'algorithme de base décrit en 4.1 est simple, et relativement facile à implémenter. L'implémentation comprend principalement quatre parties :

1) L'analyse des données en entrée

2) La génération de l'arbre de code (l'AH proprement dit)

3) La génération d'une table de code, à partir de l'arbre

4) L'itération codant chaque caractère des données d'entrée.

Bien que le lecteur trouvera une implémentation complète à l'annexe 1, nous allons donner ici quelques propositions, parfois arbitraires, de solutions.

1) L'analyse des données en entrée

Ce problème a été résolu de manière simple par comptage des caractères en entrée :

1. Initialisation de la table des fréquences à 0
2. Tant qu'il reste des données faire :

```
lire un caractère
table_fréq[caractère]:=table_fréq[caractère]+1
```

(Algorithme 4.2)

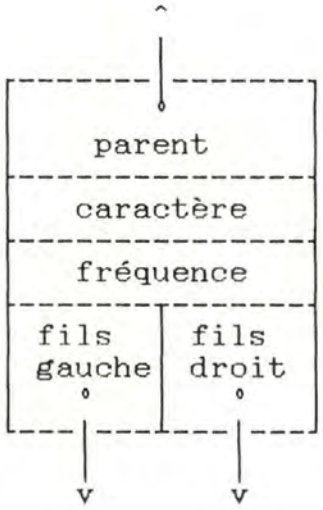
D'autres solutions existent pourtant : analyse par échantillonage, par approximation successive, tables de fréquences pré-déterminées. Nous étudierons les deux dernières respectivement au chapitre 5, et au point 4 de ce chapitre.

2) La génération de l'arbre de code

L'algorithme 4.1 génère un arbre destiné à encoder les données en entrée. Mais est-il préférable de l'utiliser directement, ou bien de générer à partir de cet arbre une table de code ? Et, à partir de ce choix, quelle devra-être la structure de l'arbre ?

Nous avons choisi la seconde solution, à la fois pour des raisons de performance et de facilité d'implémentation que nous expliciterons par la suite.

La première approche, proposée par Amsterdam, se base sur un arbre dont les éléments ont la structure suivante :

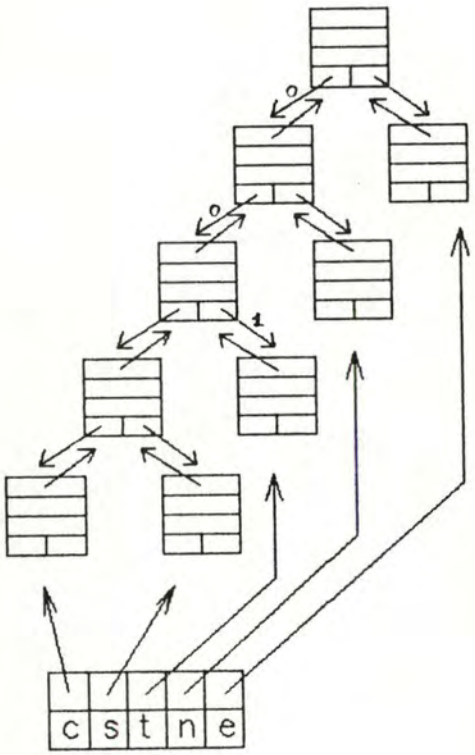


(Figure 4.2)

Sur une telle structure, le pointeur vers le parent est utilisé pour la compression, tandis que les pointeurs vers les enfants sont utilisés pour la décompression. Un tableau de pointeurs est utilisé pour l'accès aux feuilles de l'arbre.

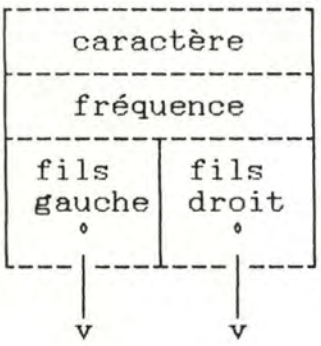
Durant la compression, l'arbre est utilisé de la manière suivante :

Soit 't' le caractère à coder ; par l'index, nous accédons à la feuille 't', et nous remontons jusqu'à la racine, en mémorisant le chemin parcouru qui est DGG, ou 100 ; nous inversons ensuite le code (001), qui est écrit dans les données de sortie.



(Figure 4.3)

La structure que nous avons utilisée est la suivante :



(Figure 4.4)

Il est à noter que l'absence du pointeur "parent" nous empêche d'utiliser cet arbre directement pour l'encodage, ce qui nous oblige à passer par l'étape intermédiaire de création d'une table des codes, explicitée au paragraphe suivant.

Une structure additionnelle, destinée à optimiser l'étape 2) de l'algorithme 4.1, est un tableau de pointeur LP destiné à maintenir un ordre dans la liste L des noeuds non encore sélectionnés. Un tel tableau aura la structure :

$\forall$ i,j : (1 $\leq$ i $\leq$ j $\leq$ taille(LP)) => (LP[i]^.freq $\leq$ LP[j]^.freq),

avec P^ signifiant "l'objet indiqué par le pointeur P".

(expression 4.4)

Il faut noter cependant que le gain de performance réalisé grâce à une telle structure, par rapport à l'algorithme de type "Selection Sort" proposé par Amsterdam, est probablement très faible, de par la taille limitée de la liste L.

3) La génération de la table des codes

La génération de cette table est faite par un algorithme récursif d'exploration complète de l'arbre, de type "en profondeur d'abord" (DFS), de la forme suivante :

```
Procedure encode(T:tree, Code:String of Bit, Depth:Integer);
begin
     If T is a leaf with character c
     then begin
               output(c); output(depth); output(Code)
          end
     else begin
               concat(code,0,newcode);
               encode(leftson,newcode,depth+1);
               /* on encode le sous-arbre de gauche */
               concat(code,1,newcode);
               encode(rightson,newcode,depth+1)
               /* on encode le sous-arbre de droite */
          end
end;
```

(Algorithme 4.3)

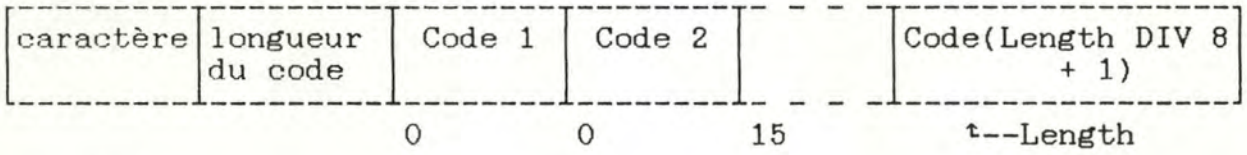
La base de cet algorithme étant fournie par la propriété Sibling : Chaque noeud ayant un Sibling, deux possibilités sont offertes : Soit il n'y a pas de fils, et nous nous trouvons à une feuille à encoder, soit il y a deux fils, et donc deux sous-arbres dont on doit déterminer le code, qui sera préfixé par la concaténation du code courant et du bit 0 pour le sous-arbre de gauche, du bit 1 pour celui de droite.

La procédure Encode est appelée initialement avec pour arguments T = Racine de l'arbre

Code = '' (code vide)

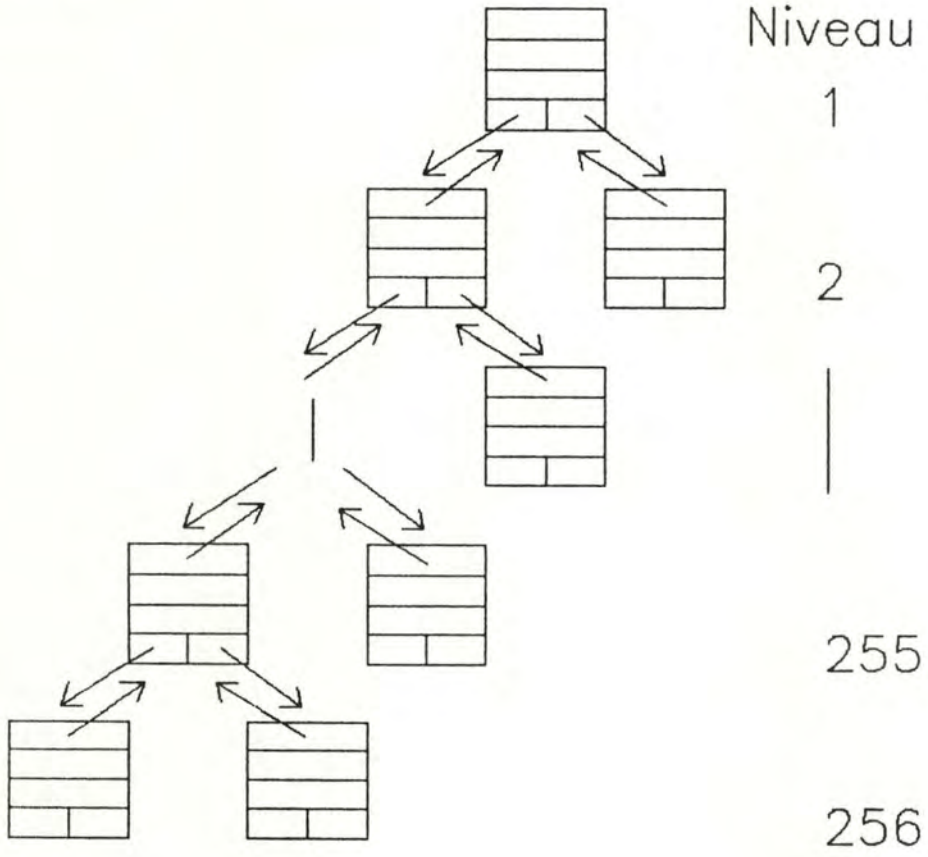
Depth = 0

La table des codes aura la structure suivante pour chaque caractère :



(Figure 4.5)

Un code aura une longueur comprise entre 1 et 256 bits, correspondant au cas le plus défavorable d'un arbre complètement dégénéré.



(Figure 4.6)

4) Le codage du caractère

Nous avons décrit au point 2) le mécanisme utilisé par Amsterdam. Lorsque l'on passe par une table de codes, ce problème est résolu de manière simple et performante, par un balayage séquentiel du code du caractère, que l'on écrit dans le buffer de sortie. Un tel mécanisme est facilement optimisable, à l'opposé d'un codage basé entièrement sur l'arbre seul.

B. La procédure de décompression

Un algorithme simple est utilisé. Il correspond à un cheminement dans l'arbre, suivant les indications données par les bits lus successivement des données comprimées (bit = 0 : aller visiter le fils de gauche ; bit = 1 : aller visiter le fils de droite) jusqu'à ce que l'on rencontre une feuille. A ce moment, on renvoie le caractère, et on se repositionne à la racine de l'arbre. Notons que préalablement au décodage proprement dit, l'arbre devra avoir été reconstruit à partir des informations de service (soit l'arbre lui-même, soit la table des codes) stockées au début des données comprimées.

Cet algorithme aura la structure suivante :

```
procedure decompresse;
begin
     Current := root;
     while not EOData do
     begin
          read a bit;
          if bit = 0
          then current := leftson
          else current := rightson;
          if current is a leaf
          then begin
                    output(character);
                    current := root
               end;
     end;
end;
```

(Algorithme 4.4)

C. Remarques

Comme que nous l'avons souligné dans ce chapitre, un code basé sur l'AH est souvent générateur d'une épargne d'espace-disque appréciable. Mais certaines faiblesses structurelles de ce schéma empêche son utilisation généralisée.

Ainsi, le système d'entrée-sortie orienté caractère des systèmes d'exploitation, entraîne un coût CPU parfois élevé, dû à la nécessité de la simulation des entrées-sorties orientées bits propres aux codes Huffman. Des problèmes de synchronisation peuvent se poser, de par la présence de données parasites dans le dernier caractère des données encodées, ce qui nous oblige à inclure dans les informations de service le nombre de caractères des données originales.

Mais la plus grande faiblesse d'un tel système est sa fragilité : la perte d'un bit des données encodées mène à la destruction complète des données suivantes. Une telle possibilité rend ce codage inutilisable dans un but d'archivage, ou plus simplement si le média magnétique (bande, disquette) n'est pas ou peu fiable. Dans cette optique, il semble qu'un mécanisme de synchronisation efficace (liste de pointeurs de resynchronisation en début des données compressées, etc.) doit encore être étudié.

4.3 La Prédiction

Une première méthode de calcul du ratio de compression à partir de la table des fréquences serait d'utiliser l'expression de Shannon :

Nombre de bits nécessaire pour l'encodage des données =

$$\sum_{i=1}^{K} [\text{nombre d'occurences du symbole Si} * - \log_2 (\text{fréquence du symbole Si})]$$

avec K = nombre de symboles de l'alphabet

(Expression 4.4)

Cette formule, permettant d'évaluer le résultat final sans construction de l'arbre des codes, ne tient malheureusement pas compte des informations de service (tables,...), ni de la redondance R des codes Huffman, définie comme étant égale à

$$R = \sum_{i=1}^{K}(Pi * Li) - H(P1,...PK)$$

$$= \sum_{i=1}^{K}(Pi * Li) + \sum_{i=1}^{K}[Pi * \log_2 (Pi)]$$

avec Pi = Probabilité d'occurence du symbole i

Li = Longueur du code Huffman du symbole i

K = Nombre de symboles de l'alphabet

H = Entropie de l'alphabet (voir [SHANNON 1948])

et calculée par Gallager comme étant

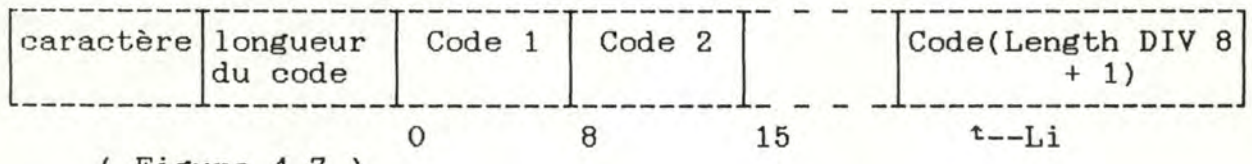
$$R \leq Pm + 0,086$$

(Expression 4.5)

avec Pm, probabilité de la lettre la plus probable de l'alphabet. [GALLAGER 1978]

L'expérience nous a montré que cette erreur de prédiction, variant avec la taille des données à comprimer, s'échelonne habituellement entre 10% et 20% de la taille des données comprimées.

Une telle variation, considérable, nous a fait préférer une méthode basée sur la table des codes, et exprimant la taille totale L(Ci) occupée par un caractère Ci dans l'ensemble des données comme étant la somme des informations utiles et de service :



(Figure 4.7)

L(Ci) = (nombre d'occurences de Ci * Longueur de son code)

+ taille de l'overhead de Ci dans la table des codes

≈ [(Ni * Li) + 16 + Li] bits

à 1/2 caractère près en moyenne

(Expression 4.6)

La taille des données compressées sera donc égale, à (K/2) caractères près en moyenne, à

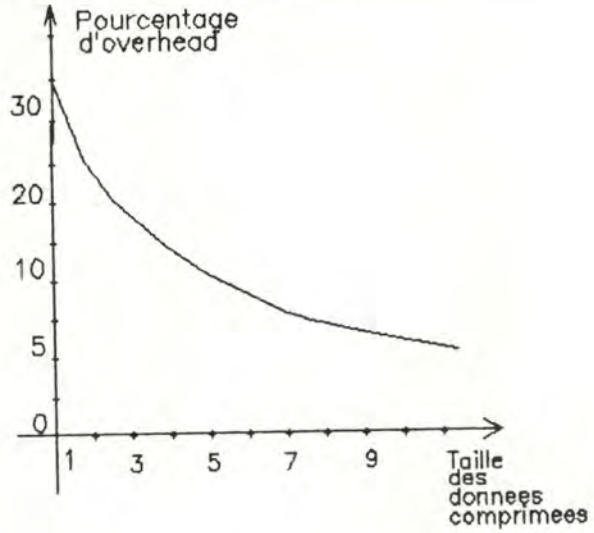
$$\sum_{i=1}^{K} (L(Ci))$$

(Expression 4.7)

4.4 Les tables Huffman statiques

Ainsi que nous l'avons observé dans ce chapitre, la taille des données codées suivant un code Huffman dévie de l'optimum hypothétique de Shannon d'une quantité égale à la somme de la redondance et des informations de services nécessaires (arbre de décodage, etc.).

Cet overhead, qui peut être regardé comme d'une taille relativement fixe, alors que la redondance est fonction croissante de la taille des données à compresser, entre pour une bonne part dans la taille de ces données, si celle-ci est peu élevée ; à cet égard, la figure 4.8 dénote un overhead pouvant aller jusque 35% des données comprimées.



(Figure 4.8)

La suppression de cet overhead, par l'utilisation de tables Huffman déterminées par analyse de programmes ou de textes (Tables COBOL, PASCAL, Anglais, ...), ou d'une table synthétique multi-usage, a été étudiée par McIntyre et Pechura [MCINTYRE 1985] à partir d'un ensemble de fichiers de programmes sources. De plus, l'utilisation de ces tables supprime la nécessité d'une

analyse préalable des données à encoder. Le coût, qui résulte en une augmentation de la redondance, reste faible lorsque les données sont de petite taille, ou d'un type bien défini.

Les résultats obtenus montrent une performance souvent meilleure que celle obtenue avec les codes Huffman traditionnels, dynamiques.

Type fichier	Taille	pourcentage de compression Huff	COB	PAS	PL/1	ALL
COBOL	18400	30,2	28,7	29,9	30,1	28,8
COBOL	17680	29,7	28,9	30,2	29,9	29,1
COBOL	23120	27,1	25,9	27,4	27,4	26,0
COBOL	20960	26,6	25,3	26,0	26,2	25,2
COBOL	32400	30,9	30,5	31,7	31,4	30,4
COBOL	56560	26,1	26,0	26,8	26,7	25,9
PL/1	32640	28,1	30,6	27,8	27,1	27,9
PL/1	32560	27,9	30,6	27,7	27,0	27,9
PASCAL	40560	27,1	28,1	26,2	26,8	26,7
PASCAL	37840	29,9	31,5	29,0	29,8	29,7

COB = table statique des fréquences dans les programmes COBOL

PAS = table statique des fréquences dans les programmes PASCAL

PL/1 = table statique des fréquences dans les programmes PL/1

ALL = table synthétisant les tables COB, PAS et PL/1

(Figure 4.9)

Notons que de telles tables sont disponibles pour les langages naturels (Français, Anglais, etc.). Un tel principe est d'ailleurs utilisé dans le code Morse.

Même si une telle méthode garde certaines lacunes : redondance élevée pour de gros fichiers, difficulté du choix de la table appropriée, elle semble une bonne alternative au codage Huffman classique.

4.5 Un Codage statistico-logique

L'algorithme Huffman, qui se base sur une vue globale des données, est habituellement inadapté à la représentation d'une base de données à accès dynamique.

Pour remédier à ce défaut, [CORMACK 1985] présente une méthode utilisée dans l'Information Management System (IMS) d'IBM, tenant à la fois du codage Run-Length, du codage statistique avec tables statiques, et de l'approche logique propre à LOVER (voir chapitre 6).

Dans cette approche, l'information d'une base de donnée est conservée sous forme d'un ensemble d'enregistrements de différents formats, chaque enregistrement pouvant être considéré comme une suite de champs d'un type spécifique.

Dans IMS, et dans un but de compression, nous pouvons considérer un champ comme appartenant à l'ensemble { caractère, chiffre, chiffre compacté (2 chiffres / octet), blanc, autre }. Pour chacun de ces types, une table Huffman statique est considérée, soit prédéterminée, soit reconstruite régulièrement. Notons que cette reconstruction, qui permet de prévenir un changement statistique dans la nature des informations, nécessite une reconstruction complète de la base de données. Enfin, le nombre de types peut être plus ou moins étendu, ce qui entraîne une adaptabilité meilleure (ainsi qu'il apparaît dans le tableau suivant), pour un coût plus élevé en tables statiques.

Types	Désignation	Bits/Char.
1		3,55
2	blanc, autre	3,23
3	blanc, alphanumérique, autre	2,92
4	blanc, alphabétique, numérique, autre	2,68
5	blanc, alphabétique, numérique, compacté, autre	2,62
8	blanc, voyelle, consonne, 0, 1-9, nul, compacté, autre	2,56

(Figure 4.10)

Le codage proprement dit s'établit de la manière suivante :

1) Coder le caractère C1, premier caractère de l'enregistrement, suivant une table choisie arbitrairement.

2) Vi, 2 ≤ i ≤ Longueur de l'enregistrement :

Coder Ci suivant la table correspondant au type du caractère C(i-1).

(Algorithme 4.5)

L'heuristique utilisée ici étant que, dans un enregistrement considéré comme une suite de champs d'une longueur de 1 caractère, le type d'un champ est habituellement le même que celui du champ précédent.

Le problème des données parasites situées dans le dernier octet de l'article, peut être résolu de manière traditionnelle (stockage de la longueur de l'article dans le(s) (deux) premier(s) octet(s) de l'article), ou par le mécanisme décrit au chapitre suivant, et consistant en la création d'un 257ème caractère que nous dénommerons EOData. Ce problème est résolu dans IMS d'une manière différente, de par l'existence d'autres contraintes, et ne nous intérèsse pas directement.

Les résultats (voir Figure 4.11) observés par Cormack sur une base opérationnelle de données administratives portent sur une réduction à la fois de l'espace disque occupé, et du nombre d'entrées-sorties physiques et doivent être rapprochés de ceux obtenus par LOVER et détaillés en chapitre 8. Le coût CPU entraîné par cette optimisation semble peu important.

	Taille globale	Entrées-Sorties physiques	CPU (Amdahl V7)
sans compression	2205 KB	1022	32,0 s.
avec compression	1275 KB	681	37,5 s.
ratios	57,9%	67,3%	117,2%

(Figure 4.11)

5. Les Codes Huffman adaptatifs

5.1 Introduction

Ainsi que nous l'avons vu au chapitre précédent, deux caractéristiques, qui sont aussi deux désavantages, de l'algorithme Huffman, sont la nécessité d'une analyse préalable des données à compresser, et une vue statistique globale de ces données.

Si certaines propositions ont été faites afin de résoudre le premier point (voir chapitre 4.4), et si la possibilité de décomposer un fichier composite (librairies, ...) en plusieurs parties homogènes plus facilement compressibles est réalisable, une solution élégante a été proposée par Gallager dans [GALLAGER 1978].

L'idée de cet algorithme se base sur un arbre de code traditionnel qui, à partir d'une situation initiale connue à la fois des processus codeur et décodeur, est modifié en fonctions du passé, c'est-à-dire des estimations statistiques courantes.

Nous allons étudier par la suite les trois problèmes se posant dans la réalisation d'un tel mécanisme, à savoir :

1) Le choix de l'arbre initial

2) Le processus de vieillissement des statistiques

3) La mise à jour de l'arbre de code

les problèmes liés au codage et au décodage proprement dits ne différant pas de ceux rencontrés dans l'implémentation de l'algorithme Huffman traditionnel.

5.2 L'Arbre initial - le Processus de Vieillissement

1) L'arbre initial

Un choix d'arbre initial adapté aux caractéristiques statistiques des données à comprimer, sans être d'une importance vitale, peut être la source d'une épargne espace et CPU intéressante. En effet, le choix d'un code fixe traditionnel de type ASCII ou EBCDIC est souvent peu adapté, et résulte donc en un codage non-optimal qui, s'il s'adapte plus ou moins rapidement à la configuration statistique des données, ne le fait qu'au prix de nombreuses modifications de l'arbre pouvant aller, dans le cas le plus grave, jusqu'à sa reconstruction complète, ainsi que l'a démontré Gallager dans [GALLAGER 1978].

De plus, le problème des données parasites dans le dernier byte (voir 4.3), nous oblige à disposer d'un caractère spécial de fin de données. Ceci est aisément réalisé en transformant un des caractères en un noeud intermédiaire, dont les fils sont le caractère transformé, et le signe spécial de fin de données. Ceci résulte en un alphabet dans lequel 255 caractères ont un code de 8 bits, et 2 caractères ont un code de 9 bits. Cette propriété des codes Huffman, à savoir l'extensibilité de l'alphabet, débouche sur des applications sortant du domaine de la compression de données. Pourtant, il faut remarquer que l'utilisation d'un code de 16 bits permettant de coder toutes les paires possibles de caractères, et utilisé par la suite comme base d'une compression Huffman traditionnelle ou adaptative, pourrait donner lieu à des taux de compression importants.

Enfin, l'utilisation de fonctionnalités de plus en plus orientées vers l'utilisateur, tels les logiciels de traitement de textes, ou l'utilisation croissante de langages informatiques évolués, ont abouti à la présence de données résidentes de plus en plus proches du langage naturel.

Pour ces raisons, un arbre initial basé sur les fréquences rencontrées dans le langage naturel semble être mieux adapté aux données traitées. Une autre possibilité serait qu'une analyse statistique préalable des caractéristiques des données résidentes globales soit effectuée, ou encore l'utilisation d'un paramètre déterminant l'arbre initial à partir d'un ensemble de choix reflétant les données rencontrées dans un système (Anglais, ASCII, COBOL, PASCAL, ...). Ce choix devrait bien entendu être reconnaissable par le processus décodeur, ce qui implique une standardisation. Même si, ainsi que nous l'avons remarqué précédemment, un choix erroné conduisait à une inefficience temporaire et à un coût temps-calcul parfois élevé, il faut noter que ces arbres de code sont souvent proches les uns des autres (voir 4.4).

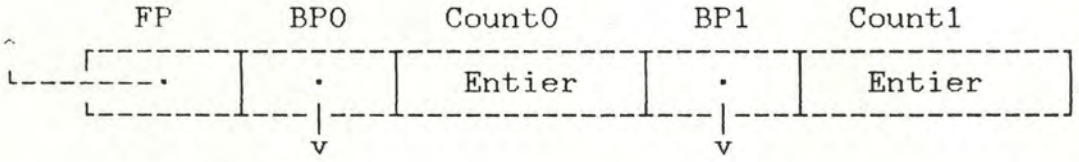
2) Les données statistiques et leur vieillissement

L'arbre initial déterminé, la mémorisation des données statistiques, représentant l'historique des données, se fait d'une manière simple par le maintien de compteurs d'occurences incrémentés chaque fois que le caractère concerné est rencontré.

Le processus de vieillissement, quant à lui, peut être traité de manières beaucoup plus variées. Le schéma original de Gallager est simple : tous les N caractères, multiplier les compteurs par un nombre fixe $\alpha < 1$. Ce schéma, dans lequel l'influence d'une occurence donnée d'un caractère sur la longueur du code est fonction inverse de son âge, est simple, et bien adapté à une source dont la configuration statistique varie lentement, si le choix de N et α est judicieux. Une valeur de N trop petite, ou un α trop grand, donnera aux interférences statistiques une influence trop grande, tandis qu'un choix inverse trop prononcé amènera une adaptabilité du code trop lente pour être encore efficace. Cormack et Horspool ont donné dans [CORMACK 1984] des algorithmes adaptés à des schémas de vieillissement plus sophistiqués, tel qu'un vieillissement à facteur exponentiel positif ou négatif.

5.3 La Structure de Données - le Codage et le Décodage

L'algorithme de Gallager se basant sur la définition structurelle des codes Huffman vue au chapitre 4, il est plus approprié d'implémenter l'arbre de code sous la forme d'une liste de paires de Siblings. Une telle paire aura la structure :



(Figure 5.1)

avec FP pointeur vers le parent des deux Siblings

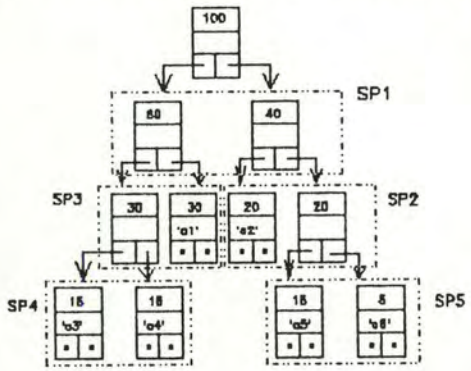
BP0 pointeur vers la paire de Siblings (les enfants) de l'élément de droite

BP1 pointeur vers la paire de Siblings (les enfants) de l'élément de gauche

Count0 fréquence du sous-arbre dont la racine est l'élément de gauche

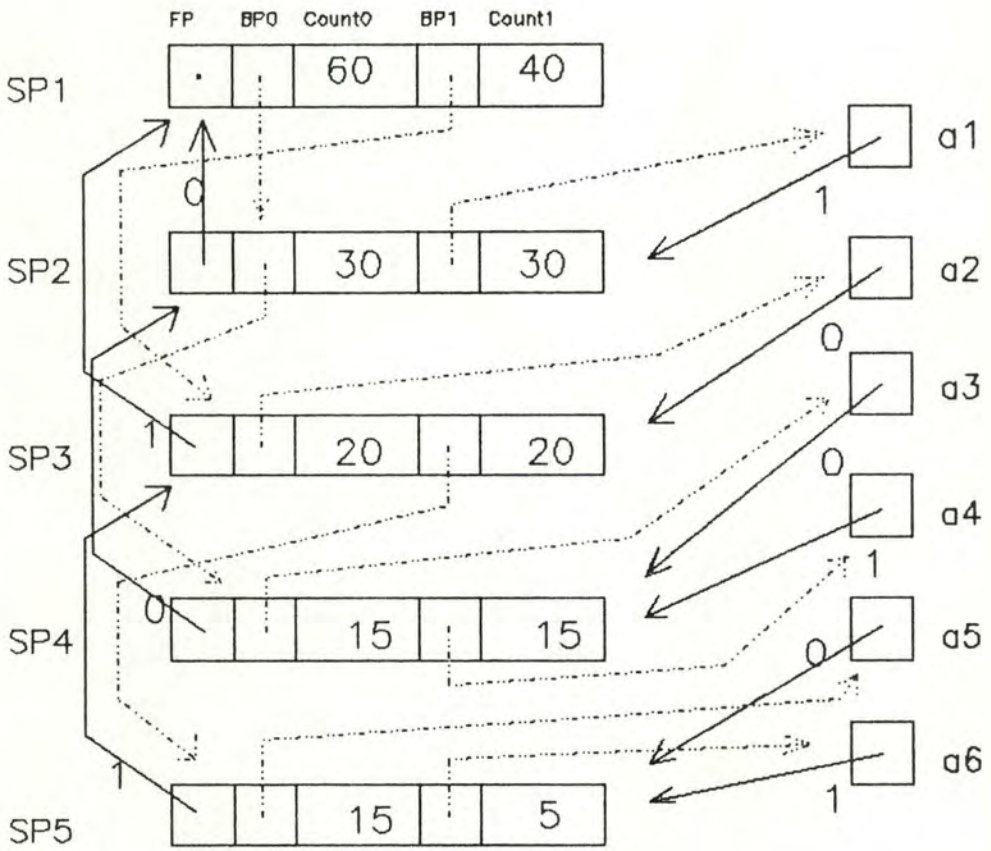
Count1 fréquence du sous-arbre dont la racine est l'élément de droite.

Ainsi, pour représenter l'arbre suivant (exprimé suivant la structure expliquée au chapitre 4) d'un alphabet {'a1'...'a6'}



(Figure 5.2)

nous aurons la structure



(Figure 5.3)

Le décodage à partir d'une telle structure est relativement aisé : ainsi, un code 001 donnera un chemin

$$SP1 \xrightarrow{0} SP2 \xrightarrow{0} SP4 \xrightarrow{1} \text{'a4'}$$

et donc le symbole 'a4'.

De même, le codage ne diffère en rien de celui proposé dans [AMSTERDAM 1986], consistant en un cheminement dans l'arbre à partir du caractère vers la racine, et en l'inversion du chemin parcouru afin d'obtenir le code à générer.

5.4 La Mise a Jour de l'Arbre

Ainsi que nous l'avons montré en 4.1, un code est optimal si et seulement si l'arbre de code correspondant possède la propriété Sibling. L'idée de base d'un tel algorithme est donc la maintenance de cette propriété, de la manière suivante :
Pour chaque caractère à coder :

1) Générer le code ; ceci doit être fait avant toute modification de l'arbre, afin de permettre au décodeur de reconnaître le code émis, et de mettre à jour son propre arbre de décodage.

2) Remonter dans l'arbre vers la racine, en incrémentant les compteurs des noeuds rencontrés les uns après les autres. Chaque fois qu'un noeud est incrémenté, son compteur est comparé avec ceux de la paire de Siblings supérieure. Si un de ces compteurs est inférieur, les deux noeuds sont échangés, ce qui correspond à permuter les sous-arbres qui ont ces noeuds pour racine.

(Algorithme 5.1)

De cette manière, la propriété Sibling est maintenue : les sous-arbres de grande fréquence sont déplacés vers le sommet de l'arbre, ou, exprimé en terme de liste de Siblings, les noeuds de grande fréquence sont déplacés vers le sommet de la liste.

6. Une Approche logique : la Méthode LOVER

Dans les méthodes étudiées dans les chapitres précédents, les données sont traitées suivant leur structure de base, à savoir une séquence non-structurée de caractères. Cette vue des données, essentielle pour les algorithmes Run-Length et Huffman, est aussi très pratique : dans certains systèmes (UNIX, MS-DOS), elle est parfois la seule disponible, toute structuration logique étant laissée à la responsabilité du programme d'application ou d'utilitaires.

Sur des systèmes plus complexes, tel le système SIEMENS BS2000, un niveau logique plus élevé est habituellement fourni : SAM (Sequential Access Method) ou ISAM (Indexed Sequential Access Method), qui restent pourtant basées sur un niveau primaire PAM (Primary Access Method) proche des entrées-sorties de base.

Nous allons étudier dans ce chapitre une méthode originale de compression, basée sur une structure logique SAM ou ISAM : la méthode LOVER (LOw Variations of succEssive Records).

6.1 Les Bases théoriques

Ainsi que nous l'avons vu dans le chapitre précédent, la probabilité d'une unité d'information émise par une source, qui est ici le groupe d'informations à compresser, est habituellement fortement dépendante des unités précédemment reçues. Cette propriété, est déjà utilisée dans les codes Huffman adaptatifs comme dans le codage Run-Length, ou la mémoire est limitée au dernier caractère reçu.

La méthode LOVER peut être regardé comme une généralisation de la méthode Run-Length, avec une mémoire de taille égale à celle de l'article logique.

En effet, on peut observer de fréquentes répétitions de valeurs d'items entre articles successifs : fichier de personnes trié sur le nom de famille, valeurs d'items limitées (Etat civil, Sexe), etc. (Remarquons encore que le terme de fichier, s'il paraît mieux adapté à une méthode se basant sur la structure en articles logiques, ne préjuge pas d'une non-applicabilité de cette méthode à des groupes de données destinés, par exemple, à transiter sur des canaux de télécommunications).

On peut donc appliquer la transformation suivante sur un fichier F, considéré comme une séquence d'articles logiques R, chacun d'eux étant considérés comme un tableau, de longueur variable, comportant au plus N caractères :

soit $F = \{ R_i[1..L_i],\ 1 \leq L_i \leq N,\ 1 \leq i \leq \#F \}$

$\forall\ (R_{i-1}, R_i) \in F,\ 2 \leq i \leq \#F$

$\forall\ j,\ 1 \leq j \leq Min(L_{i-1}, L_i)$

Si $R_{i-1}[j] = R_i[j]$

alors $R_i[j]$:= caractère spécial

avec L_i = Longueur de l'article logique i

#F = Nombre d'articles logiques du fichier F

(Algorithme 6.1)

Il est à noter que ce caractère spécial doit être choisi de telle manière qu'il n'apparaisse pas dans les données à compresser.

Le lecteur remarquera que cette transformation n'implique aucune compression. En effet, une telle méthode doit être vue comme un traitement préalable, destiné à améliorer les ratios de compression obtenus par les algorithmes Run-Length et Huffman, ainsi qu'il apparaît sur les exemples suivants :

1) extrait de programme source PASCAL

```
Begin«                                  Begin
     Tab_freq[i].freq_car:=0.0;«             Tab_freq[i].freq_car:=0.0;«
     Tab_freq[i].car:=chr(0);«          ******************car:=*h*(0);«
     Tab_freq[i].gauche:=NIL;«          ******************g*uche:=NIL*«
     Tab_freq[i].droite:=NIL«           ******************droit******«
end;«                                   end;«
```

avec '«' = "Fin de l'article" et '*' = caractère spécial

2) extrait de programme source COBOL

```
MAIN.«                                  MAIN.«
     PERFORM PART1.«                         PERFORM PART1.«
     PERFORM PART2.«                    ******************2*«
     PERFORM PART3.«                    ******************3*«
     PERFORM BOUCLE UNTIL FIN.«         *************BOUCLE UNTIL FIN.«
     EXIT PROGRAM.«                     *****EXIT PROGRAM.«
```

3) extrait d'un fichier de données administratives

```
DELARME   1963MATHEMATIQUES    «    DELARME   1963MATHEMATIQUES    «
DELHOR    1963SCIENCES         «    ***HOR *******SCIENCES      ***«
DELHORS   1964SCIENCES-POLITIQ«     ******S******4********-POLITIQ«
```

(Un extrait de base de données réelle est donné en annexe 2)

Un tel traitement, favorisant l'apparition fréquente de répétitions plus ou moins importantes de caractère spécial, semble renfermer d'importantes potentialités de compression, à la fois sur le contexte (Run-Length), et statistiquement (Huffman).

Notons que cette propriété est utilisée dans d'autres méthodes logiques de compression [STUCKY 1986].

6.2 L'Implémentation

A. Le codage

Se basant sur une vue logique de l'ensemble de données, cet algorithme doit donc y être aussi adapté. Afin de rester le plus général, nous utiliserons dans la description qui va suivre une structure de données abstraites dénommée "String", sur laquelle nous définirons les opérations suivantes :

(Remarque : le terme "Fonction" correspond à la notion de fonction Pascal ou C)

Get_String(File_in,Str,OK) : lit l'article courant dans File_in, préalablement ouvert, positionne l'indicateur EOF à TRUE si c'est le dernier article du fichier, et transfère cet article dans Str si la taille de Str est adéquate, auquel cas OK est mis à la valeur TRUE ; dans le cas opposé, Str est indéterminé, et OK est mis à la valeur FALSE.

Put_String(File_out,Str) : écrit le contenu de Str dans File_out, préalablement ouvert.

Fonction Length_String(Str) : renvoie la longueur de Str $\in$ [0..N], valant 0 si Str est vide ou indéterminé.

Set_Length_String(Str,Length,OK) : Si Length $\leq$ longueur maximale d'une donnée de type "String", alors OK est mis à la valeur TRUE, et Length_String(Str) = Length ; sinon OK est mis à la valeur FALSE, et Str n'est pas modifié.

Fonction Getcar(Str,Pos) : si Pos $\leq$ Length_String(Str), renvoie le Pos-ième caractère de Str ; sinon, le résultat est indéterminé.

Setcar(Str,Pos,Car) : si Pos $\leq$ Length_String(Str), Getcar(Str,Pos) = Car ; sinon, Str n'est pas modifié.

La procédure centrale de l'algorithme étant

Code(Str1,Str2,Caractère_spécial)

Après exécution de cette procédure, Str2 a subi la transformation décrite par la boucle interne de l'algorithme 6.1, et Str1 contient la valeur de Str2 avant transformation.

Une description de "Code" à partir des primitives décrites pourrait être :

```
Procedure Code(Str1,Str2,Car_special);
begin
     Temp:=Str2;
     for i:=1 to Min(Length_string(Str1),Length_String(Str2))
     do if (Getcar(Str1,i)=Getcar(Str2,i))
        then Setcar(Str2,i,Car_special);
     Str1:=Temp
end;
```

(Algorithme 6.2)

La boucle extérieure de l'algorithme 6.1 étant réalisée de la manière suivante :

```
Procedure LOVER_Coding(File_in,File_out);
begin
     open input File_in;
     EOF:=FALSE;
     open output File_out;
     Set_Length_String(Str1,0,OK); /* Str1 := Empty String */
     while not EOF do
     begin
          Get_String(File_in,Str2,OK);
          if not OK then Stop_error;
          Code(Str1,Str2,Car_special);
          Put_String(File_out,Str2)
     end;
     close File_in, File_out
end;
```

(Algorithme 6.3)

Si un tel algorithme doit être adapté au format des données, une implémentation relativement générale adaptée aux fichiers de type TEXTE MS-DOS est proposée en annexe 5.

B. Le décodage

Les mêmes opérations étant définies sur le type de données "String", nous pouvons définir l'opération inverse de celle réalisée par la procédure "Code" :

```
Decode(Str1,Str2,Car_special)
```

telle qu'après la séquence d'opérations

```
Temp1:=Str1;
Temp2:=Str2;
Code(Str1,Str2,Car_special);
Decode(Temp1,Str2,Car_special);
```

(Str2=Temp2) aie la valeur TRUE.

Nous pouvons définir cette procédure à partir des primitives décrites :

```
Procedure Decode(Str1,Str2,Car_special);
begin
     for i:=1 to Min(Length_String(Str1),Length_String(Str2))
     do if (Getcar(Str2,i)=Car_special)
        then Setcar(Str2,i,Getcar(Str1,i));
     Str1:=Str2
end;
```

(Algorithme 6.4)

La partie extérieure de l'algorithme de décodage étant :

```
Procedure LOVER_Decode(File_in,File_out);
begin
     open input File_in;
     EOF:=FALSE;
     open output File_out;
     Set_Length_String(Str1,0,OK); /* Str1 := Empty String */
     while not EOF do
     begin
          Get_String(File_in,Str2,OK);
          if not OK then Stop_error;
          Decode(Str1,Str2,Car_special);
          Put_String(File_out,Str2)
     end;
     close File_in, File_out
end;
```

(Algorithme 6.5)

6.3 La Prédiction

Bien qu'une telle prédiction puisse être effectuée par les méthodes vues en 3.3 et 4.3, à partir des données générées par LOVER, il faut remarquer que la structure des données, impliquant la présence d'un "overhead" difficilement accessible à LOVER (clés ISAM, pointeurs, etc.), risque d'induire des erreurs dans la prédiction des ratios de compression obtenus par l'application des algorithmes Run-Length et Huffman, ces deux méthodes opérant à un niveau plus bas, orienté caractère.

Ce n'est bien évidemment pas le cas dans un système simple de fichiers, tel MS-DOS, où toutes les données sont en général accessibles et à la charge du programme d'application.

7. Une Comparaison des Différentes Méthodes

Dans ce chapitre, nous allons examiner les trois principales méthodes de compression étudiées, et ce sous les aspects quantitatifs et qualitatifs. Notons que ces derniers ont déjà été abordés dans l'étude des méthodes proprement dites.

Dans une première partie, nous examinerons les résultats obtenus par la compression Run-Length.

Dans une seconde partie, nous examinerons le codage Huffman et nous comparerons ce codage à la méthode Run-Length. Nous étudierons aussi la combinaison de ces deux approches.

La troisième partie enfin sera consacrée à l'analyse des apports de la méthode logique LOVER aux codages précédents, seuls ou combinés.

Les tests portent sur deux ensembles distincts de données. Le premier, d'un volume de 2.36 Mégabytes, consiste en un extrait de données générées par l'utilitaire d'archivage (ARCHIVE) de BS2000. Ces données, sous la forme de 78 blocs d'une taille moyenne de 31 Kilobytes, sont non structurées. Pour cette raison, les tests portent uniquement sur les algorithmes Run-Length et Huffman et leur combinaison.

L'impossibilité de mesurer l'apport de LOVER sur de telles données, ainsi que certains résultats particuliers étonnants, nous ont amenés à approfondir les mesures sur un système MS-DOS utilisé pour le développement de logiciel. Ce second groupe de mesures a été effectué sur un ensemble de vingt-huit fichiers sélectionnés, typiques à un tel environnement, pour un volume total de 1.18 Mégabytes :

- des fichiers de textes descriptifs : memoire, geobase.hlp, chap3.mfr, *.out

- des fichiers de programmes sources Cobol, Pascal, C et Prolog : les fichiers *.cbl, *.pas, *.c, geobase.inc.

- un fichier créé par le générateur de code Forms-2 de Microfocus : selappar.dds

- une importante librairie composite : help.lib

- des fichiers binaires : draw8086.exe, adap2.pix

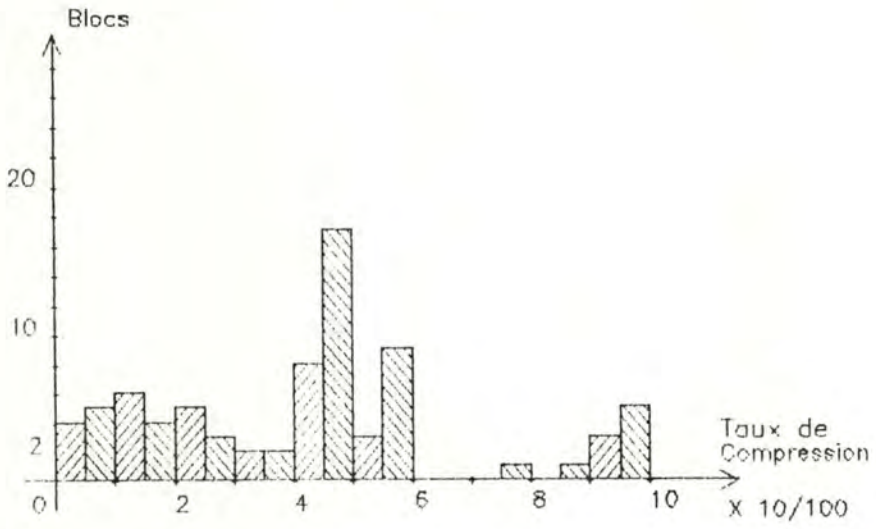
- une base de données : geobase.dba

Enfin, il faut noter que les mesures de temps effectuées, qui tiennent compte des temps d'Entrées-Sorties, ne doivent être observées qu'avec la plus grande circonspection. En effet, la première partie des tests, implémentée en PASCAL BS2000, souffre de l'absence dans ce langage de primitives de manipulation de bits. Leur simulation entraîne un facteur multiplicatif probablement situé entre 10 et 50. De plus, la seconde partie, réalisée sur un système manquant de moyens précis de mesures, souffre de la lenteur de ses Entrées-Sorties. Enfin, il faut noter que dans les deux cas l'implémentation a été réalisée dans un soucis de clarté et de modifiabilité plutôt que de performances. Des tests ultérieurs sous MS-DOS ont montré un rapport 1:40 entre la vitesse d'un algorithme orienté caractère et celle d'un algorithme tenant compte des particularités physiques du système, telle la taille des secteurs de disques.

7.1 Le Codage Run-Length

Le premier train de mesures effectué, mesures synthétisées par la figure 7.1 et détaillées en Annexe 6, laisse entrevoir des résultats particulièrement remarquables.

En effet, la compression des 78 blocs (2.36 MB) donne un résultat de 0.9 MB, ce qui correspond à un taux de compression (= "Figure of Merit") global de 41.6%. L'analyse des taux de compression de chaque bloc dénote une moyenne de 42%, et un écart-type de 25.9%.

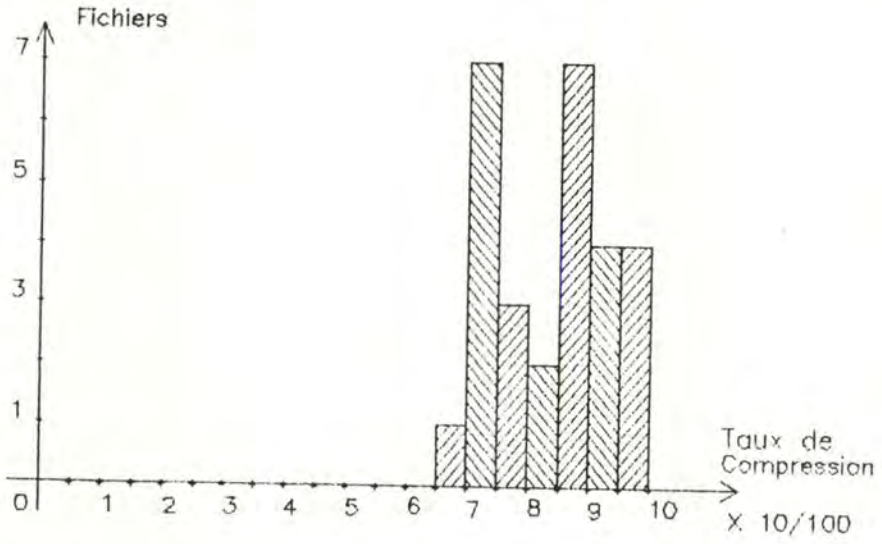


(Figure 7.1)

Malheureusement, la présence de 24 blocs, dont 19 successifs, comprimés à un taux inférieur à 25%, semble indiquer un échantillon peu significatif au niveau de la compression. Des mesures ultérieures sur un volume plus représentatif de 43.54 Mégabytes, volume constitué de 1464 blocs ARCHIVE, ont montré un taux de compression de 72.59%.

D'un point de vue temps de traitement, les mesures, qui tiennent compte des temps d'entrées-sorties sur bandes magnétiques, donnent pour l'ensemble des blocs une moyenne de 0.03 millisecondes par caractère en entrée, et de 0.14 par caractère en sortie. Les écarts-types, respectivement de valeurs 0.008 et 0.197 millisecondes, indiquent une meilleure représentativité du nombre de caractères en entrée comme indicateur du temps de traitement.

La seconde partie des mesures, synthétisée sur la figure 7.2 et détaillée dans le tableau 7.1, donne un taux de compression global plus réaliste de 82.7%, la moyenne des taux de compression par fichier étant de 84.5% pour un écart-type de 9%.



(Figure 7.2)

L'analyse des temps de traitement, qui plus encore que précédemment doivent être examinés en tenant compte des réserves émises dans l'introduction, dénote des moyennes de 0.011 secondes

par caractère en entrée, et 0.013 secondes par caractère en sortie. Les écarts-types, respectivement de 0.0008 et 0.0006 secondes, contredisent ceux observés sur le premier groupe de mesures. L'analyse de l'algorithme démontrant une complexité théorique (voir [AMSTERDAM 1985]) en $\Theta(n)$, et donc linéaire par rapport au nombre de caractères en entrée, ainsi que les remarques faites en introduction, semblent indiquer des mesures particulièrement peu fiables.

Fichier	Caract. Entrée	Taux Comp.	Temps MM-SS
MEMOIRE	286080	75%	51-14
MODULE-1.CBL	26744	90%	05-14
MODULE-2.CBL	19176	86%	03-31
MODULE-3.CBL	25844	90%	05-01
MODULE-4.CBL	25717	93%	05-06
MODULE12.CBL	45919	88%	08-48
MODULE34.CBL	51560	91%	10-06
MODULE.CBL	97478	90%	18-54
COMPHUFF.PAS	36931	74%	06-23
COMPRUN.PAS	7077	79%	01-19
CMPLOVER.PAS	2566	88%	00-30
DECOMPHU.PAS	11010	73%	01-55
DECLOVER.PAS	3196	85%	00-37
CONCAT.PAS	63628	76%	11-10
SELAPPAR.DDS	10657	91%	02-02
GEOBASE.HLP	3513	98%	00-44
GEOBASE.INC	22473	98%	04-38
GEOBASE.DBA	40935	100%	08-36
HELP.LIB	99968	94%	20-17
CHAP3.MFR	15872	87%	03-03
ADAP2.PIX	25600	67%	04-08
DRAW8086.EXE	35328	97%	07-25
WINDOWS.C	92808	75%	16-10
TEMPNRES.C	20858	82%	03-50
CHAP3.OUT	17280	73%	02-59
CHAP4.OUT	36608	75%	06-22
CHAP34.OUT	53700	75%	09-20

(Tableau 7.1 : Compression par Algorithme Run-Length)

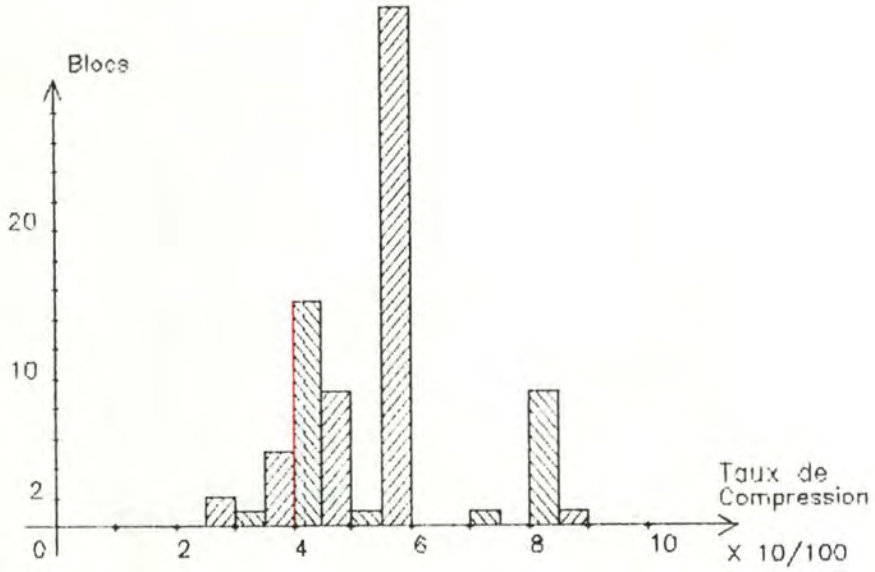
Pour synthétiser, il semble bien que le premier groupe de mesures effectuées reflète assez mal les taux de compression réalisables, tout en étant représentatif au niveau des temps de traitement. A l'inverse, les mesures effectuées sur système MS-DOS, bien que représentatives des possibilités réelles de compression de ces méthodes, le sont beaucoup moins dans les mesures des temps de traitement.

Enfin, si nous examinons les taux suivant les types de fichiers, nous pouvons remarquer que les fichiers les plus compressibles sont les fichiers images, qui comportent de fréquentes suites de caractères équivalents, les fichiers de textes descriptifs, comportant certaines séquences particulièrement compressibles de caractères (espaces, soulignés, ...), et les fichiers sources C et Pascal. En effet, ces deux langages semblent encourager l'utilisation fréquente par les programmeurs d'indentation successives dans le corps des programmes, et par là la présence de nombreuses chaînes de caractères blancs.

A l'inverse, les fichiers exécutables et les programmes sources Cobol et Prolog semblent moins compressibles, ces derniers de par leur structure proche du langage naturel et la fréquente absence d'indentation dans les programmes de ce type.

7.2 Le Codage Huffman

Ici encore, le premier train de mesures montre des résultats parfois impressionnants, avec un taux de compression global de 53.9%, une moyenne par bloc de 54.8% et un écart-type de 15.3%. Ces mesures sont détaillées en Annexe 6 et synthétisées sur la figure 7.3.

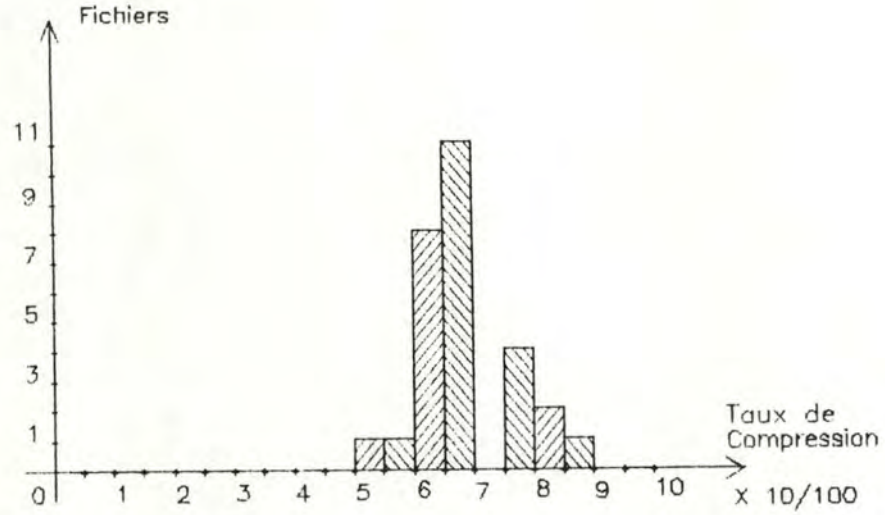


(Figure 7.3)

La présence de 33 blocs comprimés à plus de 50% montre là aussi, bien que de manière moins nette, un échantillon peu représentatif du taux de compression réalisable.

Les mesures portant sur les temps de traitement par caractère donnent en entrée une moyenne de 1.13 milliseconde et en sortie 2.14 millisecondes, avec des écarts-types respectifs de 0.14 et 0.31 milliseconde. Cette faible différence ne permet pas de tirer déjà des conclusions sur l'indicateur de complexité.

La seconde partie des mesures, détaillée dans le tableau 7.2 et synthétisée sur la figure 7.4, dénote un taux de compression moyen par fichier de 68.3%, avec un écart-type de 7.5%.



(Figure 7.4)

Les temps de traitement, en moyenne de 0.03 seconde en entrée et 0.05 seconde en sortie, ne contredisent pas dans leurs proportions les mesures réalisées sous BS2000. Il reste que le faible écart entre ces temps et ceux réalisés lors du codage Run-Length dénote à suffisance l'influence des Entrées-Sorties dans les mesures effectuées sous MS-DOS. L'analyse de l'algorithme démontre une complexité théorique $\Theta(n)$, linéaire par rapport au nombre de caractères en entrée. En effet, l'influence de la longueur variable des codes est contrebalancée par la fréquence variable des caractères, l'une étant fonction inverse de l'autre.

Enfin, les écarts importants entre les deux méthodes pourraient être en partie compensée par l'optimisation des routines de manipulation de bits nécessaires au codage Huffman.

Fichier	Entrée	Comp.	Temps
MEMOIRE	286080	60%	158-55
MODULE-1.CBL	26744	67%	15-07
MODULE-2.CBL	19176	68%	10-52
MODULE-3.CBL	25844	64%	14-30
MODULE-4.CBL	25717	68%	14-44
MODULE12.CBL	45919	65%	25-57
MODULE34.CBL	51560	68%	29-13
MODULE.CBL	97478	66%	54-55
COMPHUFF.PAS	36931	63%	20-38
COMPRUN.PAS	7077	68%	04-00
CMPLOVER.PAS	2566	82%	01-31
DECOMPHU.PAS	11010	67%	06-13
DECRUN.PAS	2848	77%	01-39
DECLOVER.PAS	3196	81%	01-52
CONCAT.PAS	63628	65%	35-45
SELAPPAR.DDS	10657	70%	06-00
GEOBASE.HLP	3513	76%	02-03
GEOBASE.INC	22473	76%	13-04
GEOBASE.DBA	40935	65%	23-05
HELP.LIB	99968	67%	56-29
CHAP3.MFR	15872	75%	09-09
ADAP2.PIX	25600	61%	14-05
DRAW8086.EXE	35328	88%	11-06
WINDOWS.C	92808	54%	50-28
TEMPNRES.C	20858	68%	11-51
CHAP3.OUT	17280	62%	09-36
CHAP4.OUT	36608	62%	20-23
CHAP34.OUT	53700	58%	29-34

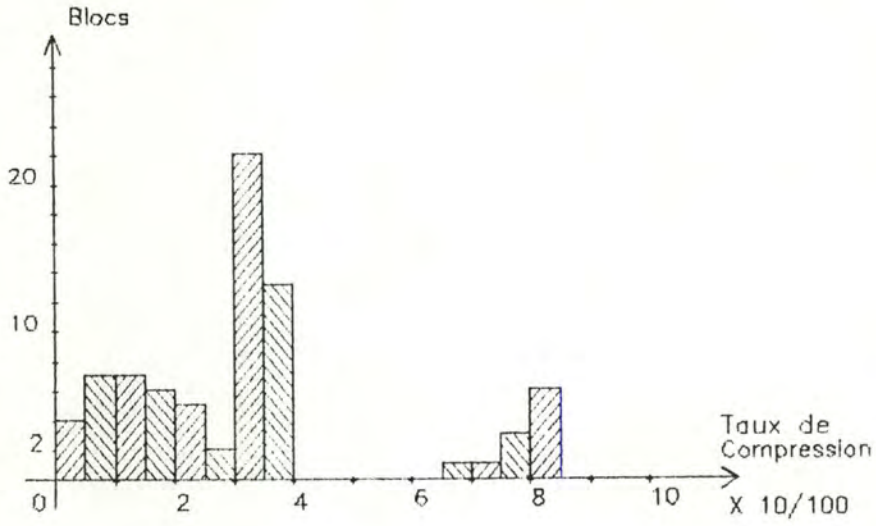
(Tableau 7.2 : Compression par Algorithme Huffman)

La_combinaison_des_codages_Run-Length_et_Huffman

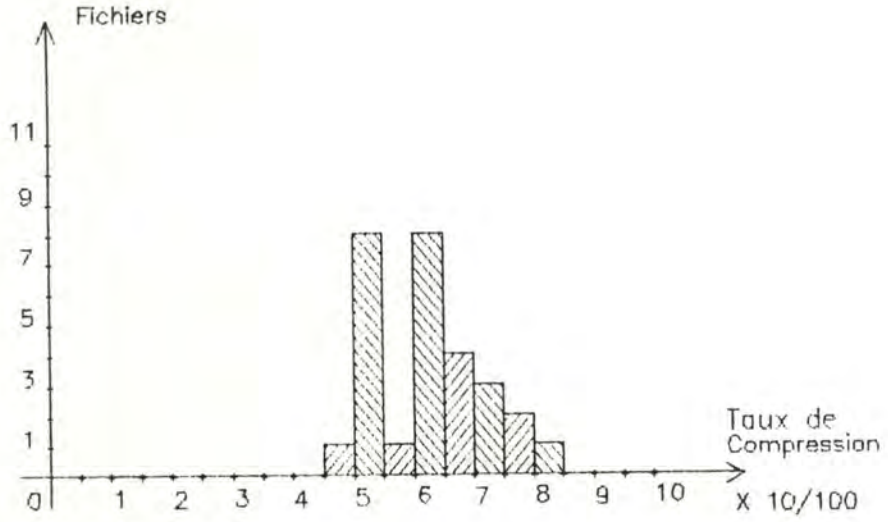
L'utilisation combinée des codages Huffman et Run-Length se justifie par deux propriétés de ce dernier évoquées dans son étude théorique. En effet, le codage Run-Length, basé sur le contexte, ne compresse généralement que des chaînes particulières (blancs, cadres de tableaux, ...). De plus, ces chaînes ont

elles-mêmes des longueurs qui ne sont pas réparties uniformément sur l'intervalle [4..255]. Un codage différenciant ces chaînes à la fois sur leur type et leur longueur, tout en tenant compte des éléments non compressés préalablement (fréquence des lettres dans le langage naturel) ou introduits par la compression (caractère spécial), peut donc amener une compression additionnelle appréciable.

Les mesures effectuées, exposées en Annexe 6 et sur la Figure 7.5 pour les mesures sous BS2000, dans le tableau 7.4 et sur la Figure 7.6 pour celles réalisées sur MS-DOS, montrent dans les deux cas des résultats appréciables avec des compressions moyennes respectivement de 32.7% et 62.9%.



(Figure 7.5)



(Figure 7.6)

Fichier	Entrée	Comp.	Temps
MEMOIRE	286080	52%	122-26
MODULE-1.CBL	26744	64%	13-47
MODULE-2.CBL	19176	60%	09-22
MODULE-3.CBL	25844	63%	13-16
MODULE-4.CBL	25717	63%	13-30
MODULE12.CBL	45919	61%	23-04
MODULE34.CBL	51560	62%	20-37
MODULE.CBL	97478	62%	49-47
COMPHUFF.PAS	36931	54%	15-46
COMPRUN.PAS	7077	77%	03-17
CMPLOVER.PAS	2566	77%	01-20
DECOMPHU.PAS	11010	55%	04-40
DECRUN.PAS	2848	69%	01-19
DECLOVER.PAS	3196	72%	01-36
CONCAT.PAS	63628	55%	27-40
SELAPPAR.DDS	10657	67%	05-33
GEOBASE.HLP	3513	74%	01-59
GEOBASE.INC	22473	73%	12-47
GEOBASE.DBA	40935	66%	23-02
HELP.LIB	99968	63%	52-57
CHAP3.MFR	15872	67%	07-59
ADAP2.PIX	25600	51%	09-54
DRAW8086.EXE	35328	86%	20-33
WINDOWS.C	92808	53%	40-03
TEMPNRES.C	20858	61%	09-54
CHAP3.OUT	17280	52%	07-15
CHAP4.OUT	36608	51%	15-37
CHAP34.OUT	53700	50%	22-42

(Tableau 7.3 : Compression par Algorithme combiné)

Note : ces temps ne tiennent compte que de la compression Huffman

Plus important que ces mesures absolues, la comparaison avec les ratios obtenus par l'utilisation isolée des deux méthodes est probant :

	Huff.	R-L	R-L + Huf.
BS2000	54.8%	42.0%	32.7%
MS-DOS	64.2%	84.6%	62.9%

(Tableau 7.4 : Comparaison des Taux moyens de Compression)

Les temps de traitement dénotent eux aussi une amélioration de la combinaison des deux méthodes par rapport à la méthode Huffman seule. En effet, l'algorithme Huffman, principal consommateur de ce temps de traitement, traite moins de caractères en entrée. L'écart est dans le cas du premier groupe de mesures particulièrement significatif, avec une moyenne de 1.13 milliseconde/caractère en entrée pour le codage Huffman seul, et une moyenne de 0.59 milliseconde/caractère en entrée pour le codage combiné.

Il semble donc que l'utilisation d'un tel algorithme combiné résulte à la fois en une augmentation du taux de compression, et en une diminution du temps de traitement principal nécessaire à cette compression. Ceci nous amènera à développer un tel algorithme combiné dans le chapitre 8.

7.3 Le Codage LOVER

Cette technique, essentiellement destinée à l'archivage de fichiers de données, a été utilisée pour le traitement de l'ensemble des fichiers constituant le deuxième échantillon.

Les résultats sont décrits dans le tableau 7.5, et synthétisés dans les Figures 7.7 à 7.9 respectivement pour les combinaisons de LOVER avec les algorithmes Run-Length, Huffman, et la combinaison de ces deux derniers.

Fichier	Entrée	LOVER +R-L	R-L	LOVER +Huf	Huf	LOVER +R-L +Huf	R-L +Huf	
MODULE-1.CBL	26744	71%	90%	60%	67%	51%	64%	°
MODULE-2.CBL	19176	58%	86%	61%	68%	44%	60%	°
MODULE-3.CBL	25844	70%	90%	63%	64%	50%	63%	°
MODULE-4.CBL	25717	70%	93%	61%	68%	61%	63%	°
MODULE12.CBL	45919	65%	88%	58%	65%	58%	61%	°
MODULE34.CBL	51560	69%	91%	61%	68%	50%	62%	°
MODULE.CBL	97478	68%	90%	60%	66%	48%	62%	°
COMPHUFF.PAS	36931	75%	74%	65%	63%	55%	54%	
COMPRUN.PAS	7077	78%	79%	72%	68%	75%	77%	
CMPLOVER.PAS	2566	86%	88%	85%	82%	77%	77%	
DECOMPHU.PAS	11010	75%	73%	70%	67%	57%	55%	
DECRUN.PAS	2848	76%	78	78%	77%	68%	69%	
DECLOVER.PAS	3196	84%	85%	79%	81%	72%	72%	
CONCAT.PAS	63628	76%	76%	64%	65%	55%	55%	
SELAPPAR.DDS	10657	52%	91%	62%	70%	45%	67%	°
GEOBASE.HLP	3513	97%	98%	77%	76%	75%	74%	
GEOBASE.INC	22473	92%	98%	74%	76%	70%	73%	
GEOBASE.DBA	40935	75%	100%	60%	65%	52%	66%	°
HELP.LIB	99968	94%	94%	66%	67%	63%	63%	
CHAP3.MFR	15872	86%	87%	73%	75%	67%	67%	
WINDOWS.C	92808	69%	75%	56%	54%	49%	53%	
TEMPNRES.C	20858	78%	82%	65%	68%	59%	61%	
CHAP3.OUT	17280	72%	73%	60%	62%	53%	52%	
CHAP4.OUT	36608	73%	75%	61%	62%	50%	51%	
CHAP34.OUT	53700	73%	75%	63%	58%	50%	50%	

(Tableau 7.5 : Compression par combinaison avec LOVER)

L'utilisation de LOVER montre une amélioration moyenne respectivement de 6.2%, 1.8% et 8.5%. L'examen détaillé du

Tableau 7.5 montre que ce gain s'effectue essentiellement (°) :

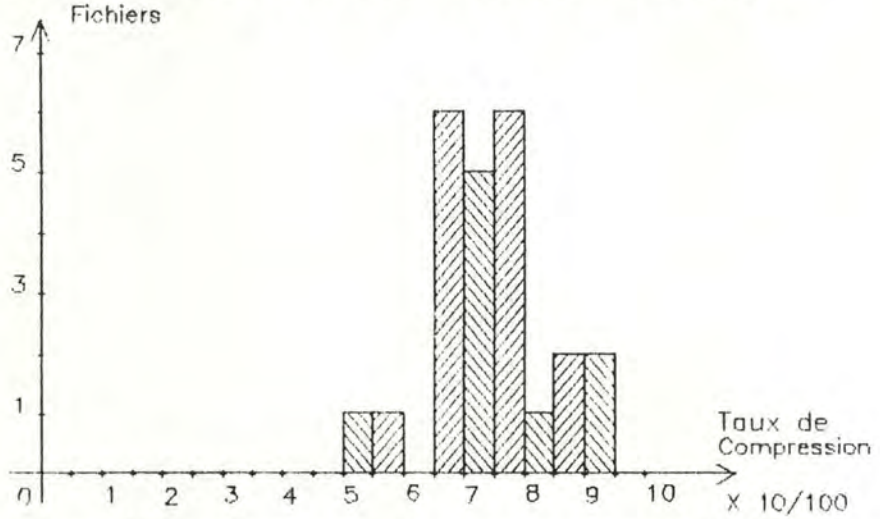
- sur les programmes sources Cobol, probablement de par la présence de successions de phrases et de déclarations alignées

- sur le fichier créé par le générateur, de par l'alignement des déclarations sur certains mots-clés (FILLER, PIC, ...)

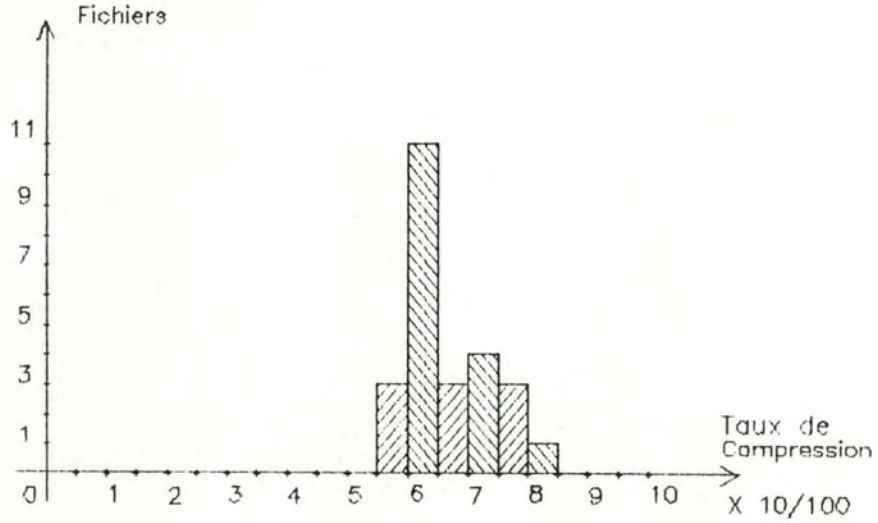
- sur la base de données, des extraits codés de cette dernière étant reproduits en Annexe 2.

Il faut noter les gains remarquables obtenus sur les deux derniers, gains confirmés par des études ultérieures réalisées sur des fichiers générés par le générateur de code Forms-2, et sur deux bases de données utilisées dans des applications pharmaceutiques et immobilières.

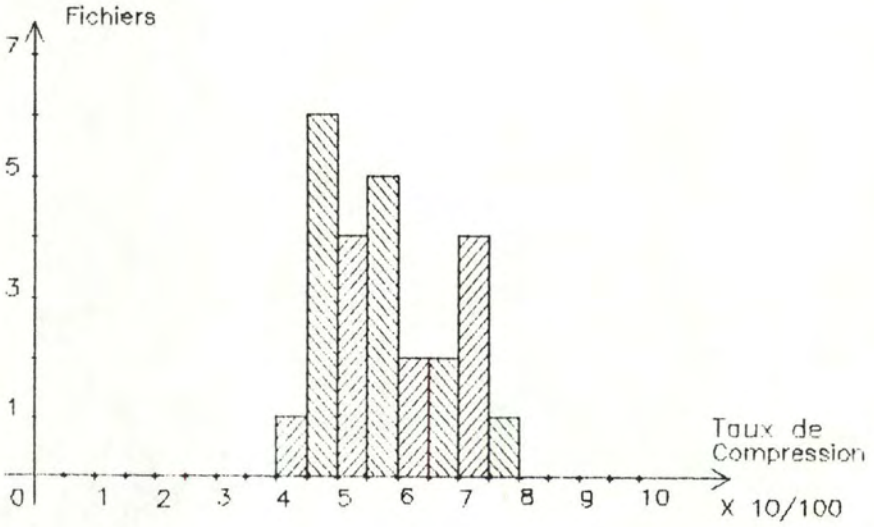
Les performances en termes de temps de traitement n'ont pas été étudiées ici. En effet, elles varient fortement suivant le langage utilisé et le niveau logique des Entrées-Sorties. Ainsi, le codage LOVER du fichier "Geobase.dba" nécessite 6'15" par l'algorithme décrit en Annexe 5, temps réduit à 9 secondes après optimisation des buffers et utilisation d'ordres d'Entrées-Sorties adaptés à la structure logique du fichier.



(Figure 7.7 : Moyenne de compression LOVER + Run-Length)



(Figure 7.8 : Moyenne de Compression LOVER + Huffman)



(Figure 7.9 : Moyenne de Compression LOVER + Run-Length + Huffman)

7.4 Conclusion

Sur un plan uniquement quantitatif, il semble bien que chacune des méthodes proposées aie ses avantages et ses inconvénients. Ainsi, la méthode Run-Length donne des taux de compressions généralement modestes, bien que soumis à des variations spectaculaires. Le coût d'une telle méthode est pourtant si faible que son utilisation la plus systématique possible est à conseiller. Il semble d'ailleurs qu'une commande de compression de type Run-Length, optimisée, et n'effectuant pas d'analyse préalable des données, puisse être réalisée avec des performances approchant les commandes traditionnelles de copie de fichier.

Le codage Huffman présente un potentiel de compression généralement important et stable. Sa complexité et son coût élevé peuvent être un inconvénient majeur dans certaines applications. Là aussi, il semble pourtant qu'une optimisation des parties critiques de l'algorithme, à savoir les primitives de manipulation et d'Entrées-Sorties de bits, puisse résoudre en partie ce problème.

Mais une leçon importante de ces tests est que le codage Huffman seul se justifie rarement : en effet, les mesures détaillées montrent que sur 106 groupes (blocs et fichiers) d'informations, 105 sont compressés avantageusement par un algorithme combinant les deux approches, et ce à un coût plus faible. Un algorithme spécifique implémentant cette combinaison sera donc proposé au chapitre suivant.

Enfin, même si l'utilisation d'un algorithme vertical tel LOVER semble ne trouver sa pleine utilité que pour des données bien structurées, son utilisation peut apporter un gain appréciable sur d'autres types de données, tels des programmes sources ou des textes descriptifs.

Ainsi, il semble que la combinaison de méthodes se basant sur des propriétés différentes des données puisse apporter des gains importants. A cet égard, l'utilisation d'autres techniques (remplacement de mots-clé, ...) pourrait-elle aussi apporter un gain substantiel aux trois algorithmes étudiés.

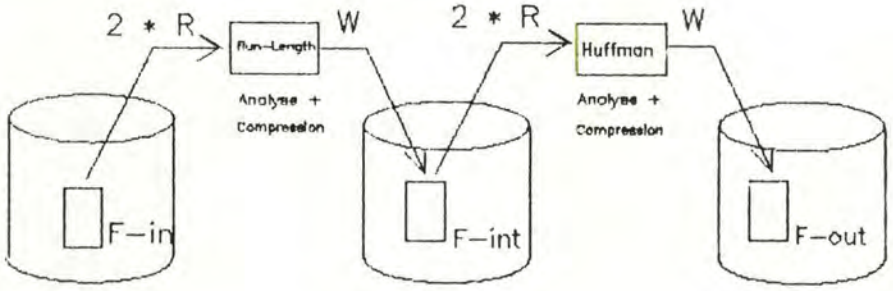
Sur un plan plus qualitatif, l'algorithme Run-Length possède de nombreux avantages : sa simplicité permet une implémentation Assembler pouvant encore diminuer son coût déjà très faible. A l'opposé, l'algorithme Huffman est très coûteux, et une implémentation optimisée dans un langage de type Assembler est rendue difficile par sa complexité. Mais le plus grand problème de cet algorithme, déjà évoqué précédemment, est la fragilité du code généré, qui le rend inapplicable sur des médias peu fiables en l'absence de codes auto-correcteurs.

Enfin, LOVER est simple et facilement adaptable. Cette adaptabilité et son orientation logique de haut niveau permet de l'implanter aisément dans une application manipulant des fichiers à des fins d'archivage.

8. Un Algorithme combiné pour la Compression et la Décompression

8.1 Introduction

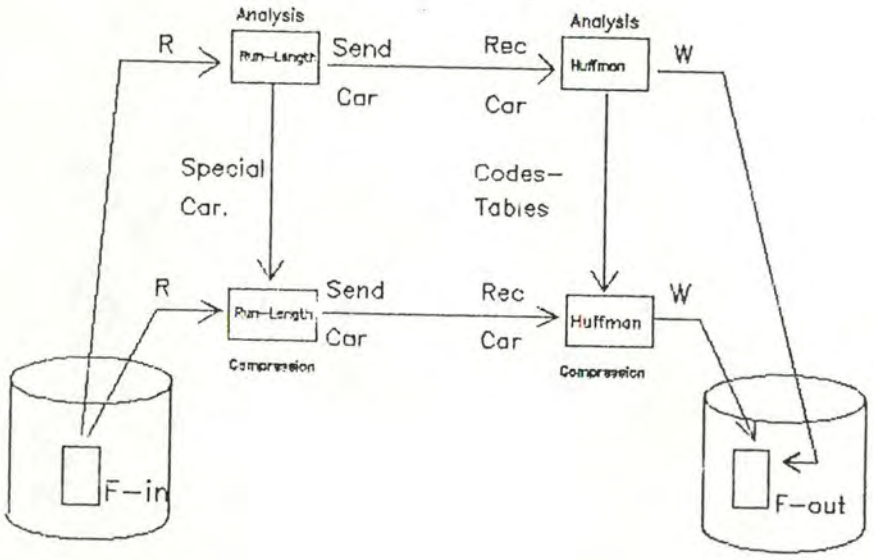
Ainsi que nous l'avons remarqué dans le chapitre 7, l'utilisation consécutive d'algorithmes de compression se basant sur des caractéristiques différentes des données est fréquemment efficiente, à la fois en terme de volume de données généré, et en temps de calcul. Un tel algorithme combiné, ainsi que nous l'avons utilisé dans le chapitre 7, pourrait être représenté graphiquement de la manière suivante :



(Figure 8.1)

Malheureusement, si la réalisation d'un tel mécanisme est immédiate, l'étape intermédiaire, consistant en une écriture et deux lectures du fichier intermédiaire, est très coûteuse. La suppression complète du fichier intermédiaire est réalisable, moyennant certaines adaptations compliquant à la fois la réalisation et la modifiabilité du processus.

Ces modifications, effectuées à la fois sur un plan horizontal (mode de communication entre processus successifs intervenant dans le flot de données) et sur un plan vertical (modifications des processus eux-mêmes), sont synthétisées sur la figure 8.2.



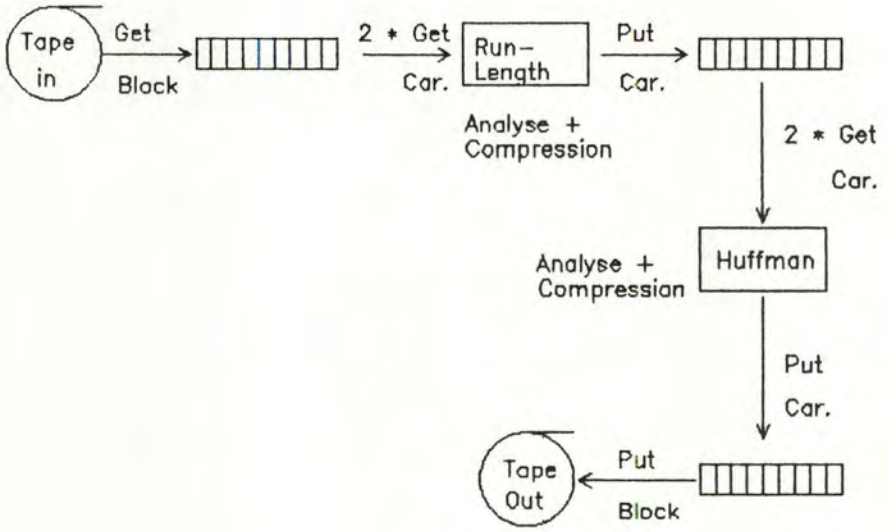
(Figure 8.2)

Les questions posées par l'utilisation d'un tel algorithme combiné, qui épargne à la fois l'espace disque nécessaire au fichier intermédiaire, et le temps parfois important nécessaire aux entrées-sorties additionnelles, concernent essentiellement le mécanisme utilisé pour la réalisation des procédures "Send-car" et "Receive-car", et les stockage des informations de service (tables de codes Huffman, caractère spécial utilisé par la compression Run-Length, taille du fichier original, ...) générées par les algorithmes d'analyse des données. Nous en proposerons plusieurs solutions aux paragraphes suivants.

Une dernière méthode, à la fois efficace, élégante, et facile à mettre en oeuvre, a été utilisée par l'auteur, dans le cadre d'une étude comparative d'algorithmes de compression, pour la firme SIEMENS A.G. (Munich, R.F.A.). Les données, générées par

l'utilitaire ARCHIVE (archivage de fichier sur bande), présentaient la particularité d'être constituées de blocs d'une taille (variable) inférieure à 32 KBytes, qui autorisait le traitement d'un bloc comme un "fichier virtuel", chacun de ces "fichiers" étant implémenté sous forme d'un tableau de caractères.

La petite taille de chaque bloc autorisant la constitution de données intermédiaires, et la nature de ces blocs autorisant un accès sans entrées-sorties, et donc à faible coût CPU, permettent la réalisation du mécanisme de base décrit à la figure 8.1 sans modifications importantes :

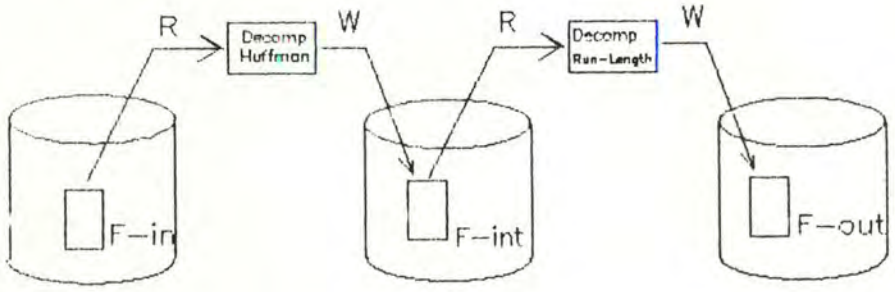


(Figure 8.3)

Le petit nombre d'entrées-sorties nécessaires, la simplicité et l'adaptabilité (facilité d'adjonction de routines supplémentaires, ...) de cette solution est contrebalancée par une utilisation intensive de la mémoire principale, une limitation de la taille des données à compresser, et une mauvaise portabilité face à des systèmes (MS-DOS) ou langages (Turbo-Pascal) ne disposant que d'un faible espace adressable.

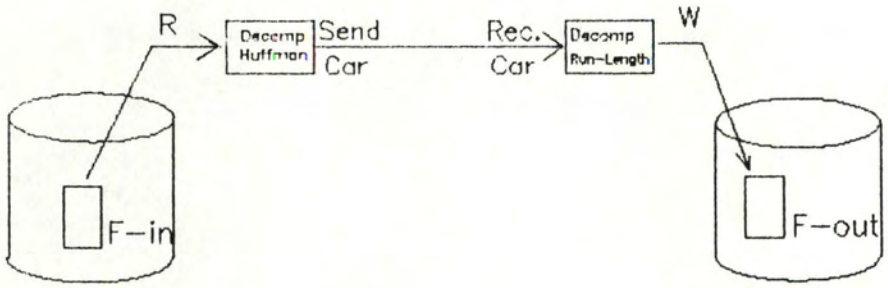
Face à de telles difficultés, l'éventualité d'une décomposition en blocs de taille inférieure, toujours réalisable, est malheureusement soumise au phénomène d'une proportion croissante d'overhead, proportion pouvant amoindrir les ratios de compression, ainsi que cela a été vu en 4.4.

Le problème de la décompression peut être résolu de manière plus simple, ceci étant principalement dû au fait que les deux algorithmes ne nécessitent qu'une seule lecture du fichier source. De l'algorithme combiné de base :



(Figure 8.4)

est dérivé, par simple suppression du fichier intermédiaire :

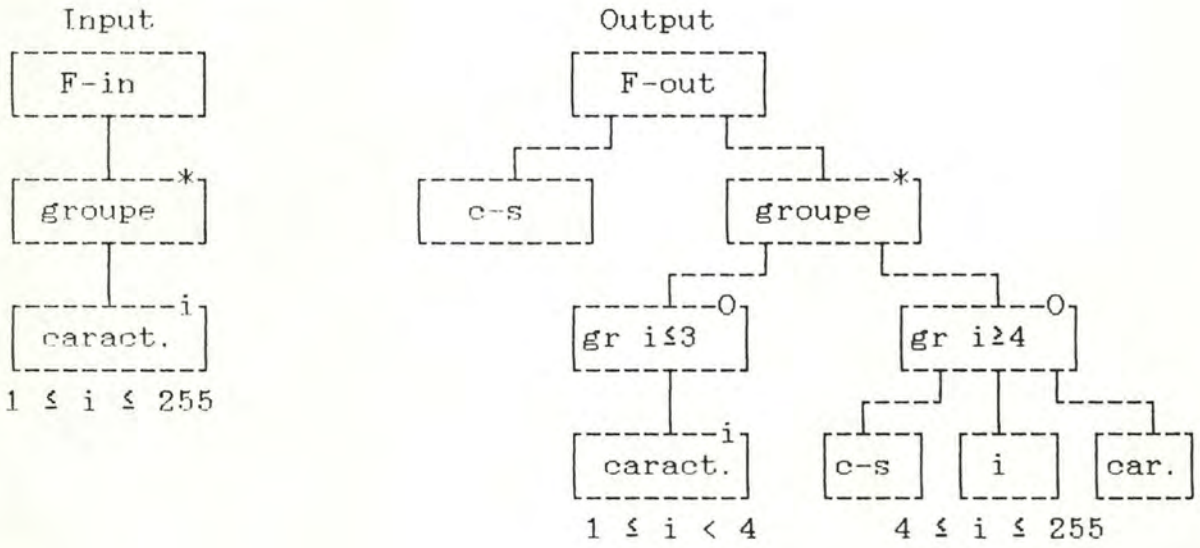


(Figure 8.5)

8.2 L'analyse et la prédiction

A partir des algorithmes de choix du caractère spécial à utiliser dans la compression Run-Length (chapitre 3), de prédiction des ratios de compression pour les méthodes Run-Length et Huffman (chapitres 3 et 4), et de la structure de base des données manipulées par les deux algorithmes, et exprimée suivant les conventions de Jackson (voir [JACKSON 1975] et chapitre 3) :

Run-Length



c-s : caractère spécial utilisé par Run-Length

Huffman

overhead : information de service générée par l'algorithme de compression (tables, ...)

(Figure 8.6)

la prévision du ratio de compression obtenu par l'algorithme combiné est effectuée à l'aide de l'algorithme vu en 4.3, appliqué sur une table fréquentielle générée par l'algorithme d'analyse Run-Length, de la manière suivante :

1) Initialiser les fréquences à 0;

2) Taut que pas fin des données faire :
   Lire un groupe de caractère C
   Si ce groupe a une longueur L < 4
   alors fréquence(C):=fréquence(C) + L
   sinon fréquence(C):=fréquence(C) + 1
         fréquence(L):=fréquence(L) + 1
         fréquence-c-s:=fréquence-c-s + 1

3) Déterminer le caractère spécial à utiliser par Run-Length par l'algorithme 3.6.

4) fréquence(car-spécial):=fréquence(car-spécial) +
                           fréquence-c-s

(Algorithme 8.1)

l'étape 4), et l'utilisation d'un registre spécial "fréquence-l-s" étant rendues nécessaires de par l'inconnue subsistant sur le choix du caractère spécial lors de la constitution de la table des fréquences.

Notons encore que la possibilité d'une distorsion, provenant du choix d'un caractère spécial présent dans des groupes de longueur inférieure à quatre caractères, et reflétant un ensemble de données probablement peu compressibles, est peu importante et ne doit normalement influencer que légèrement l'optimalité du code Huffman généré.

8.3 La décompression

A. Une implémentation parallèle

Dans un système de type UNIX, l'implémentation est aisément réalisée au sein du Shell (langage de commande UNIX), par la commande :

Decompresse_Huffman < File-in ¦ Decompresse_Run-Length > File-out

(Algorithme 8.2)

avec File-in : fichier préalablement compressé par application successive des algorithmes de compression Run-Length et Huffman.

File-out : fichier résultant de la décompression successive par les algorithmes dse décompression Huffman et Run-Length.

Decompresse_Huffman : réalisant la décompaction d'un fichier compressé suivant un algorithme Huffman

Decompress_Run-Length : réalisant la décompression d'un fichier compressé suivant un algorithme Run-Length.

Ces deux programmes devant recevoir leurs données à partir de l'entrée standard (dans UNIX, STDIN, normalement affecté au clavier du terminal), et transférant les données vers la sortie standard (dans UNIX, STDOUT, normalement affecté à l'écran utilisateur). Cette contrainte permet de rediriger le flux d'entrée à partir d'un fichier de données au moyen de la commande '<', et le flux de sortie vers un fichier au moyen de la commande '>', la communication entre les deux processus s'établissant au moyen de la création d'un "pipe" (ordre ¦), mécanisme établissant un transfert automatique de données (avec

synchronisation) entre la sortie standard du premier processus, et l'entrée standard du second.

Malheureusement, la génération par les processus de compressions de caractères reconnus par UNIX comme indicateurs d'états particuliers des entrées et sorties (notamment le caractère ASCII 027, utilisé par UNIX comme indication de fin de fichier, et généré par Run-Length lors de la compression d'une chaîne de 27 caractères) demande la création de routines particulières évitant la génération de ces caractères (décomposition d'une chaîne de 27 caractères en deux chaînes, utilisation de la technique du "Bit-Stuffing" [TANENBAUM] en cas de génération par l'algorithme Huffman du caractère ASCII 027) qui complexifie grandement la réalisation.

Au sein d'un système tel le SIEMENS BS2000, le parallélisme, implémentable, dans un langage de haut niveau tel PASCAL, au moyen de coroutines communicantes, ou en langages orientés système (Assembler, SPL4) au moyen de tâches communicant à l'aide de sémaphores ([CROCUS 1981], entraîne l'utilisation de routines systèmes coûteuses (approximativement 3000 instructions par changement de tâche) gérant ce parallélisme.

Pour ces raisons de coût et de complexité, l'implémentation séquentielle d'un tel algorithme combiné semble s'avérer plus réaliste.

B. Une implémentation séquentielle

La structure de base d'un tel algorithme combiné, de type "producteur-consommateur" ([CROCUS 1981]), est facilement dérivée de sa représentation fonctionnelle : en effet, l'implémentation traditionnelle UNIX-like de cet algorithme combiné

$$a_1 < \text{File-in} \mid a_2 \mid \ldots \mid a_n > \text{File-out}$$

(Expression 8.1)

avec a_i, $1 \leq i \leq n$, algorithme simple de compression

peut être représentée fonctionnellement par

$$\text{File-out} := f_n\,(f_{n-1}\,(\ldots(f_2\,(f_1\,(\text{File-in})))\ldots))$$

(Expression 8.2)

avec f_i, $1 \leq i \leq n$, fonction décrivant la relation entre le flux d'entrée de l'algorithme a_i, et son flux de sortie.

à partir d'une telle représentation, la structure de base d'un tel algorithme est celle du dernier algorithme appliqué et sa structure interne celle du précédent, ceci s'appliquant de manière récursive jusqu'au premier algorithme appliqué. Ainsi, l'algorithme 8.2 aura une représentation fonctionnelle

```
File-out:=Decompresse_Run-Length(Decompresse_Huffman(File-in))
```

(Expression 8.3)

ce qui correspond algorithmiquement à l'algorithme de base de la décompression Run-Length, additionné des instructions

nécessaires à la lecture et à la reconstruction des informations de service (arbre de décodage, etc.) stockées par l'algorithme de compression Huffman. Dans cet algorithme de base, la "lecture" d'un groupe de caractères (algorithme 3.4) effectue les appels nécessaires à une procédure allant lire et décompresser un caractère comprimé par l'algorithme Huffman.

```
Procedure Read_a_Group;

Begin
     If not EOF(File_in)
     then begin
               décompresse_car_huff(File_in,Car);
               if (Car=Special_char)
               then Begin
                         décompresse_car_huff(File_in,car);
                         décompresse_car_huff(File_in,Length);
                    end
               else length:=1
          end
     else End_File:="T";
end;
```

(Algorithme 8.3 - Algorithme 3.4 modifié)

8.4 La compression

La nécessité d'une première analyse des données à compresser et d'une mémorisation des informations de service nous interdit une solution purement Shell, telle celle énoncée en 8.3.

Cette mémorisation, réalisable par l'utilisation de petits fichiers temporaires ou de fichiers EAM (Evanescent Access Method, fichiers temporaires, non-structurés, à accès rapides sous BS2000), ne solutionne pas les problèmes de base déjà rencontrés en 8.3, à savoir la complexité et le coût d'une solution purement parallèle.

L'algorithme séquentiel se base ici aussi sur une procédure de compression de type Run-Length, augmentée des instructions nécessaires à la construction et au stockage de l'arbre de code généré à partir de la table des fréquences constituée par l'algorithme 8.1. La procédure d'"écriture" d'un groupe de caractère est modifiée de telle manière qu'elle fasse les appels nécessaires à une procédure codant les caractères suivant l'arbre de code préalablement constitué, ce qui ne porte que sur la procédure "Process_group" de l'algorithme 3.2

```
Procedure Process_group;
Begin
     Oldcar:=Car;
     Length_Run:=0;
     Write(File_out,Special_Char);
     While ((Car=Oldcar) & (Length_Run<255) & (End_File<>"T"))
     do Process_char;
     if ((Oldcar=Special_char) or (Length_Run >= 4)
     then compress_huff(File_out,Special_Char);
          compress_huff(File_out,Oldcar);
          compress_huff(File_out,Length_Run)
     else for i:=1 to Length_Run
          do compress_huff(File_Out,Oldcar);
end;
```

(Algorithme 8.4 - Algorithme 3.2 modifié)

9.Quelques Considérations sur l'Emplacement dans le Système

L'emplacement de routines de compression dépend essentiellement de leur mode d'utilisation : sont-elles utilisées pour l'amélioration des performances d'un système de télécommunications, ou pour diminuer le volume utilisé par les données résidentes d'un système d'information ? Le mécanisme doit-il ou non être transparent à l'utilisateur ?

9.1 Les Télécommunications

Suivant la description de l'architecture d'un réseau informatique donnée dans [TANENBAUM 1981], un tel service, qui n'est pas essentiel à son bon fonctionnement, trouve sa place au niveau "Présentation" (Couche 6) d'un logiciel de réseau.

Ceci permet d'éviter à la fois des séquences de compression et décompression successives lors du traitement des données par les hôtes intermédiaires du réseau, qui se produiraient dans le cas d'une implantation aux niveaux 1 à 3, et les inévitables problèmes de compatibilité apparaissant dans un réseau hétérogène.

Cela ne signifie nullement une implémentation purement software : si un algorithme de type Run-Length est usuellement simple et efficient lorsque réalisé sous forme logicielle, un tel algorithme peut être réalisé facilement de manière câblée.

Un algorithme de type Huffman (classique ou adaptatif), de par sa relative complexité, n'est quant à lui aisément implémentable que dans un langage de haut niveau, ce qui peut nuire à son efficience, de par le manque habituel d'instructions

de ces langages donnant accès aux niveaux les plus élémentaires des informations. Un tel algorithme câblé est pourtant réalisable, ainsi que le montre l'étude de faisabilité réalisée par Yakovleff, et présentée en Annexe 1. Notons encore qu'un algorithme Huffman adaptatif, de par son caractère filtre (pas de première passe sur les données à compresser, pas d'information de service générée) se prête particulièrement bien à ce genre d'implémentation.

9.2 Les Données résidentes

Ici aussi, ce service, qui n'est pas essentiel, et est d'ailleurs absent dans de nombreux systèmes, peut apporter des gains appréciables dans le stockage des données.

Dans un système de base de données, l'utilisation d'un algorithme de type Run-Length, si elle est aisée à implémenter, risque de ne pas donner des résultats aussi probants qu'un codage de type vertical, orienté vers la structure logique du fichier, tel l'algorithme Huffman modifié utilisé dans l'Information Management System (IMS) d'IBM, et décrit en 4.5. Un tel algorithme peut être utilisé dans une base de données de manière entièrement transparente à l'utilisateur. Un algorithme plus simple, tel LOVER, ne peut quant à lui être utilisé que comme préparation à un archivage utilisant d'autres techniques de compression, son caractère de dépendance cascadée des records empêchant toute décompaction non-séquentielle. Lui aussi peut être implanté au niveau des procédures d'archivage de la base de données.

Même s'il semble désirable pour des procédures d'archivage de disposer de routines de compression, et sans en faire nécessairement une partie du système d'exploitation, il semble que cette facilité devrait être fournie à l'utilisateur lui-même, sous forme de modules utilisables par des programmes d'application ([STUCKY 1986]), ou sous forme d'une commande directement appelable au niveau de l'interpréteur de commandes ([PECHURA 1982]), ainsi qu'il est implémenté dans la version 7 du système UNIX de Berkeley.

En effet, l'utilisateur, qui demeure en dernier ressort celui qui connaît le mieux la nature de ses données résidentes, souvent statiques (programmes sources, procédures Batch, rapports), peut alors sélectionner celles devant faire l'objet d'une compression diminuant le volume global utilisé et ce, dans un environnement où les mémoires de masse sont fréquemment restreintes, au bénéfice de tous les utilisateurs.

10. Conclusion

A la fin de cette étude apparaissent à la fois des réponses aux diverses questions posées en introduction et de nouveaux problèmes méritant eux aussi une étude appropriée.

En effet, ainsi que nous l'avons établi dans le chapitre 7, il semble que l'utilisation d'algorithmes de compression puisse apporter une dimension supplémentaire, en terme d'espace disponible, à un système de stockage de données.

Pourtant, nous ne pourrions conseiller l'implémentation systématique de telle procédures : leur efficacité dépend par trop de la nature des données. De plus, une compression faible impliquant généralement des temps de compression et décompression élevés, l'efficience de tels mécanismes varie fortement. Enfin, des problèmes d'intégrité de données peuvent se poser sur des médias magnétiques peu fiables.

Il semble dès lors que la décision de l'utilisation de procédures de compression ressorte du concepteur du système d'information lui-même, plutôt que son utilisation systématique à travers le système d'exploitation. En outre, ainsi que le note Tanenbaum, "... tout ce qui peut être enlevé du système d'exploitation doit l'être" [TANENBAUM 1981].

Deux remarques doivent cependant être faites :

1) Une technique simple, telle l'algorithme Run-Length, semble devoir être utilisée chaque fois que cela est possible et notamment dans des problèmes d'archivage et de transmission, en raison de sa robustesse, de son potentiel de compression généralement satisfaisant et de son faible coût CPU, tant lors de la compression que de la décompression.

2) Un ensemble de routines sophistiquées de compression devrait être fourni à l'utilisateur, afin de lui permettre de diminuer le volume de ses données résidant sur des mémoires rapides. Des algorithmes plus sophistiqués, tel l'algorithme Huffman et ses variantes, peuvent être combinés avec les algorithmes simples de type Run-Length. En effet, les problèmes d'intégrité des données sont ici moins critiques, de par la qualité des supports magnétiques.

En outre, ces mécanismes, généralement simples, puisent leur performance dans leur adaptation aux données à compresser. Il semble donc bien qu'ici particulièrement une compréhension profonde des bases logiques de ces algorithmes et une connaissance précise de la nature des données soient les clés de la performance. Cette parfaite maîtrise permettra l'adaptation et la modification d'algorithmes connus, pour des gains parfois importants.

Une alternative à cette maîtrise serait la création d'un système analytique intelligent, permettant de déterminer, par analyse de la structure des données, l'algorithme ou l'ensemble d'algorithmes à appliquer afin d'obtenir une compression satisfaisante. Un tel système automatique de compression pourrait utiliser des méthodes orientées vers la signification syntaxique des données ([STORER 1982], [KATAJAINEN 1986]), tel le remplacement de mots-clés, et apparentées aux techniques de compilation.

L'étude d'algorithmes "verticaux" semble aussi dénoter de remarquables possibilités. Ainsi, l'algorithme statistico-logique étudié en 4.5 peut être utilisé efficacement sur un ensemble non-

structuré de données, tel un texte descriptif.

Enfin, l'utilisation généralisée de l'algorithme Huffman semble soumise à deux contraintes : la suppression de la première lecture des données et la solution des problèmes d'intégrité des données sur les médias peu fiables généralement utilisés pour l'archivage. Le premier problème, qui pourrait être résolu par l'utilisation des codes Huffman adaptatifs étudiés au chapitre 5, implique la recherche de processus efficaces de vieillissement des statistiques. Le second problème, enfin, ne semble pas avoir rencontré à ce jour de solution satisfaisant le difficile équilibre entre sécurité et espace utilisé.

11. Bibliographie

11.1 Livres

[KNUTH 1973-1 et -3] KNUTH, D.E., The Art Of Computer Programming
Vol.1 (Fundamentals Algorithms), Addison-Wesley, 1973
Vol.3 (Sorting and Searching), Addison-Wesley, 1973

[JACKSON 1975] JACKSON, M.A., Principles of Program Design, New-York : Academic Press, 1975

[AHO 1977] AHO, A.V., and ULLMAN, J.D., Principles Of Compiler Design, Addison-Wesley, 1977

[HIGGINS 1979] HIGGINS, D.A., Program Design And Construction, Prentice-Hall, 1979

[CROCUS 1981] CROCUS (Nom collectif), Systèmes d'exploitation des ordinateurs, Bordas, Paris, 1981

[TANENBAUM 1981] TANENBAUM, A.S., Computer Networks, Prentice-Hall, 1981

[MACCHI 1983] MACCHI, C., et GUILBERT, J.-F., Téléinformatique, Bordas, Paris, 1983

[HELD 1983] HELD, G., Data Compression - Techniques and Applications, Hardware and Software Considerations, John Wiley & Sons, 1983

[SEDGEWICK 1983] SEDGEWICK, R., Algorithms, Addison-Wesley, 1983

11.2 Articles

[SHANNON 1948] SHANNON, C.E., "A Mathematical Theory of Communication", The Bell System Technical Journal, Vol. 27 N°3, July 1948, pp. 379-423 and Vol. 27 N°4, Octobre 1948, pp. 623-655

[WILLIAMS 1964] WILLIAMS, J.W.J., "Heapsort", Communications of the ACM, Vol. 7 N°6, Juin 1964, pp.347-348

[FLOYD 1964] FLOYD, R.W., "Treesort 3", Communications of the ACM, Vol. 7 N°12, Décembre 1964, p.701

[MARRON 1967] MARRON, B.A., et De MAINE, P.A.D., "Automatic Data Compression", Communications of the ACM, Vol. 10 N°11, Novembre 1967, p.711

[SNYDERMAN 1970] SNYDERMAN, M., et HUNT, B., "The Myriad Virtues of Text Compaction", DATAMATION, Vol. 16 N°16, Décembre 1970, pp. 36-40

[RUTH 1972] RUTH, S.S., et KREUTZER P.J., "Data Compression for Large Business Files", DATAMATION, Vol. 18 N°9, Septembre 1972, pp. 62-66

[HAGAMEN 1972] HAGAMEN, W.D., LINDEN, D.J., LONG, H.S., et WEBER, J.C., "Encoding verbal information as unique numbers", IBM Systems Journal, Vol. 11 N°4, 1972, pp. 278-315

[WELLS 1972] WELLS, M., "File Compression Using Variable Lengths Encoding", The computer Journal 15-4, 1972, pp. 308-313

[WAGNER 1973-1] WAGNER, R.A., "Common Phrases and Minimum-Space Text Storage", Communications of the ACM, Vol. 16 N°3, Mars 1973, pp. 148-152

[WAGNER 1973-2] WAGNER, R.A., "Algorithm 444 - An Algorithm for Extracting Phrases in a Space-Optimal Fashion", Communications of the ACM, Vol. 16 N°3, Mars 1973, pp. 183-185

[HAHN 1974] HAHN, B., "A New Technique for Compression and Storage of Data", Communications of the ACM, Vol. 17 N°8, Août 1973, pp. 434-436

[PERL 1975] PERL, Y., GAREY, M.R., et EVEN, S., "Efficient Generation of Optimal Prefix Codes for Equiprobable Words Using Unequal Cost Letters", Journal of the ACM, Vol. 22 N°2, Avril 1975, pp. 202-214

[VISVALINGAM 1976] VISVALINGAM, M., "Indexing with Coded Deltas-A Data Compaction Technique", Software-Practice and Experience, Vol. 6, 1976, pp. 397-403

[RUBIN 1976] RUBIN, F., "Experiments in Text File Compression", Communications of the ACM, Vol. 19 N°11, Novembre 1976, pp. 617-623

[ZIV 1977] ZIV, J., et LEMPEL, A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. IT-23 N°3, Mai 1977, pp. 337-343

[BOYER 1977] BOYER, R.S., et MOORE, J.S., "A Fast String Searching Algorithm", Communications of the ACM, Vol. 20 N°10, Octobre 1977, pp. 762-772

[HIGGINS 1977] HIGGINS, D.A., "Structured Programming With Warnier-Orr Diagrams", Byte, Décembre 1977, pp. 104-110, et Janvier 1978, pp. 146-151

[ZIV 1978] ZIV, J., "Coding Theorems For Individual Sequences", IEEE Trans. on Information Theory, IT-24(4), Juillet 1978, pp. 405-412.

[GALLAGER 1978] GALLAGER, R.G., "Variations On A Theme By Huffman", IEEE Trans. on Information Theory, IT-24(6), Novembre 1978, pp. 668-674.

[RODEH 1981] RODEH, M., PRATT, V.R., et EVEN, S., "Linear Algorithm for Data Compression via String Matching", Journal of the ACM, Vol. 28 N°1, Janvier 1981, pp. 16-24

[CORBIN 1981] CORBIN, H., "An Introduction to Data Compression", BYTE, Avril 1981, pp. 218-250

[BERGLAND 1981] BERGLAND, G.D., "A guided Tour of Program Design Methodologies", Computer, Octobre 1981, pp. 13-37

[CORTESI 1982] CORTESI, D., "An Effective Text-Compression Algorithm", BYTE, Janvier 1982, pp. 397-403

[TROPPER 1982] TROPPER, R., "Binary-Coded Text", BYTE, Avril 1982, pp. 398-413

[PECHURA 1982] PECHURA, M., "File Archival Techniques Using Data Compression", Communications of the ACM, Vol. 25 N°9, Septembre 1982, pp. 605-609

[STORER 1982] STORER, J.A., et SZYMANSKI, T.G., "Data Compression via Textual Substitution", Journal of the ACM, Vol. 29 N°4, Octobre 1982, pp. 928-951

[SEVERANCE 1983] SEVERANCE, D.G., "A Practitionner's Guide to Data Base Compression - Tutorial", Informations Systems, Vol. 8 N°1, 1983, pp. 51-62

[RISSANEN 1983] RISSANEN, J., "A Universal Data Compression System", IEEE Transactions On Information Theory, Vol. IT-29 N°5, Septembre 1983, pp. 656-664

[LANGDON 1983] LANGDON, G., and RISSANEN, J., "A Double-Adaptive File Compression Algorithm", IEEE Transactions On Communications, Vol. COM-31 N°11, Novembre 1983, pp. 1253-1255

[STORER 1983] STORER, J.A., "Toward An Abstract Theory Of Data Compression", Theoretical Computer Science N°24, 1983, pp. 221-237

[CORMACK 1984] CORMACK, G.V., et HORSPOOL, R.N., "Algorithms for Adaptive Huffman Codes", Informations Processing Letters, Vol. 18 N°3, Mars 1984, pp. 159-165

[CLEARY 1984] CLEARY, J., et WITTEN, I., "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions On Communications, Vol. COM-32 N°4, Avril 1984, pp. 396-402

[GONZALEZ 1985] GONZALEZ SMITH, M.E., and STORER, J.A., "Parallel Algorithms for Data Compression", Journal of the ACM, Vol. 32 N°2, Avril 1985, pp. 344-373

[MCINTYRE 1985] MCINTYRE, D.R., and PECHURA, M.A., "Data Compression Using Static Huffman Code-Decode Tables", Communications of the ACM, Vol. 28 N°6, Juin 1985, pp. 612-616

[AMSTERDAM 1985] AMSTERDAM, J., "An Analysis of Sort", BYTE, Septembre 1985, pp. 105-112

[CORMACK 1985] CORMACK, G.V., "Data Compression on a Database System", Communications of the ACM, Vol. 28 N°12, Décembre 1985, pp. 1336-1342

[STUCKY 1986] STUCKY, W., and STORK, H.-G., "GUTACHTEN über Möglichkeiten der Anwendung von Datenkomprimierung-Verfahren, speziell des Frankenstein-Lidzba-Verfahrens", unclassified document, SIEMENS internal use, Février 1986.

[FLAMCALL 1986] FLAM Schnittstelle, unclassified document, SIEMENS internal use, Février 1986.

[AMSTERDAM 1986] AMSTERDAM, J., "Data Compression with Huffman Coding", BYTE, Mai 1986, pp. 99-108

[KATAJAINEN 1986] KATAJAINEN, J., PENTTONEN, M., et TEUHOLA, J., "Syntax-directed Compression of Program Files", Sotware - Practice and Experience, Vol. 16 N°3, Mars 1986, pp. 269-276

Annexe 1 : Hardware Feasibility Study of a Huffman Algorithm

André Yakovleff, Master of Sciences in Electrical Engineering of
California Institute of Technology, Pasadena (CA.)
Current Address : Dep. KWS UP 222
SIEMENS A.G.
Munich (West-Germany)

Foreword

The purpose of this paper is to study the feasibility of the hardware implementation of a compression algorithm, called the Huffman algorithm, the principle of which is described in chapter 4 of this memoir.

The input and output data are variable length memory blocks of a maximum size of 32 KBytes.

We have chosen to discuss a solution which involves the use of both custom designed chips and general purpose RAMS and Stacks, rather than the exclusive use of existing components.

The diagrams that are presented here constitute the rough outlines of building blocks that can be - or already have been - laid out in other applications. When necessary, some details are shown concerning certain tricks which may be important for the reader's understanding.

The design should be implemented using a double phase non overlapping clock scheme. Clock phases are referred to as "Ph1" and "Ph2".

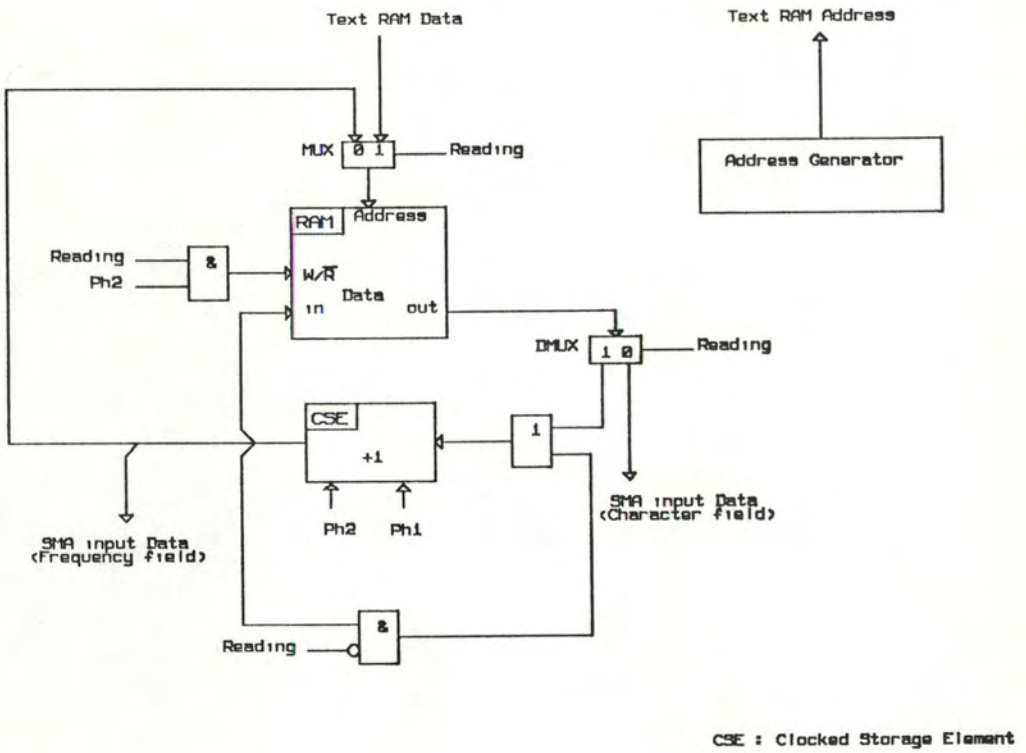


Figure 1 - Occurence Counter

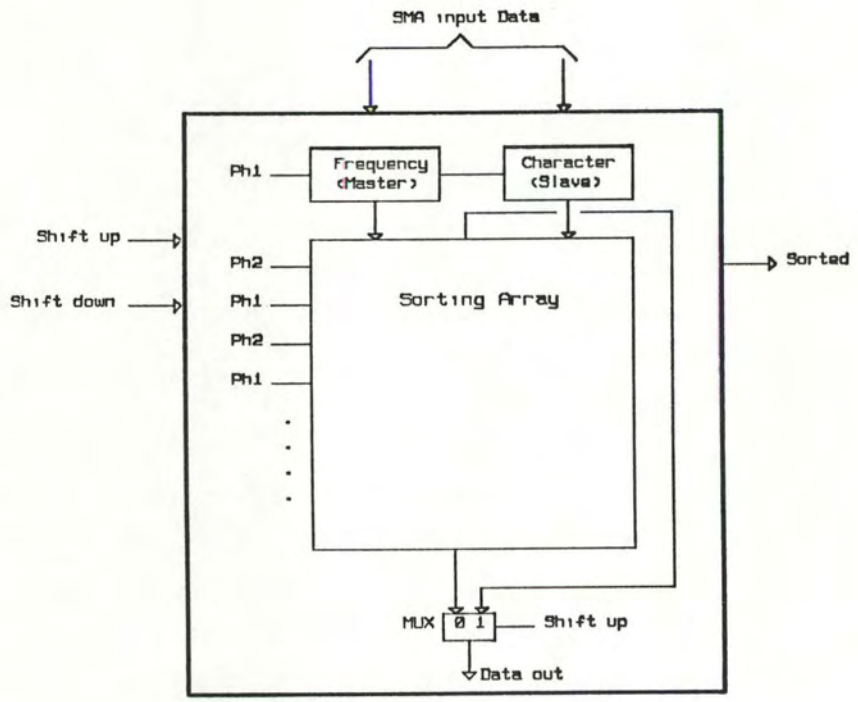


Figure 2 - Sorting Memory Array

Step 1 - Counting the number of occurrences of each character

Let us first consider the data to be compressed ; as it is held in RAM, a simple way of summing the occurrences of each character is by reading the RAM sequentially (the address being provided by the "Text Address Generator", which is merely a counter), byte by byte, using the data as the address of another RAM. The data held in the latter is then incremented and stored back. A block diagram is shown in figure 1.

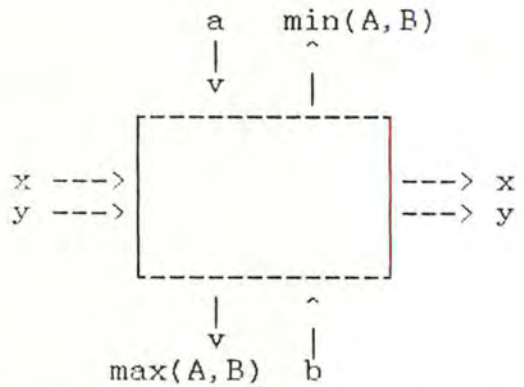
At the end of this operation, the total number of occurrences of a character "x" will be held at location "x". Notice that the CSE (Clocked Storage Element) - incrementer group is used both for counting the occurrences in Step 1 and for addressing the RAM is Step 2 (Sorting).

Step_2_Sorting_by_number_of_occurrences

There exists of course many sorting algorithms, though few of them combine minimal execution time and silicon area as efficiently a bubble sorting. Though it is unattractive in software applications, one of its main features in hardware is to achieve some degree of parallelism, which cannot otherwise be attained in any software implementation using a general purpose machine. Typically, while sorting, comparisons causing data to be shifted either way take place concurrently and involve all the data present in the array at any given time.

The sorting memory array shown in figure 2. comprises the following elements :

- A bitwise comparator with the following characteristics :

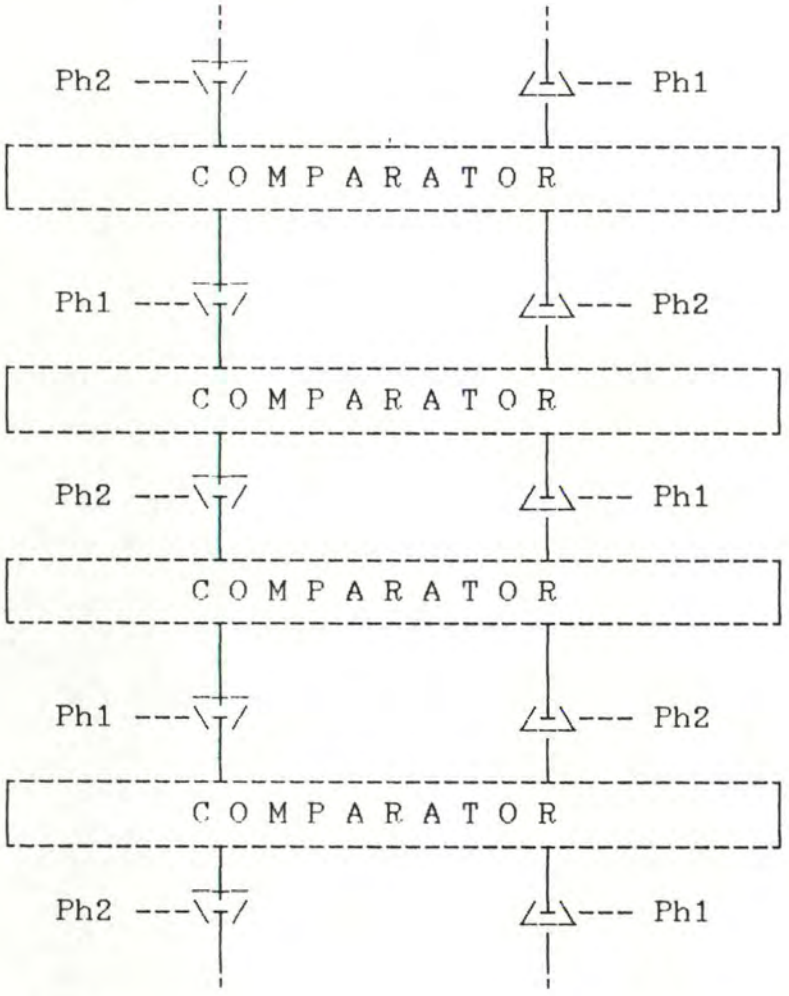


A = {a} B {b}

x	y	
0	0	A=B so far
1	0	A>B
0	1	A<B
1	1	(should not occur)

The serial connection of such elements produces a comparator of any desired width. Also a "slave" comparator can be derived from the above (one in which x and y serve merely to redirect a and b).

- Connection cells between comparators :



Without going into further details, let us note that a variety of signals such as "sort finished" and "array full" can be derived from the (X,Y) pairs at the end of each row of comparators, and with a small amount of additional logic in the connection cells, control signals such as "shift up" or "shift down" can be easily implemented, this time using the (x,y) pairs at the beginning of each row.

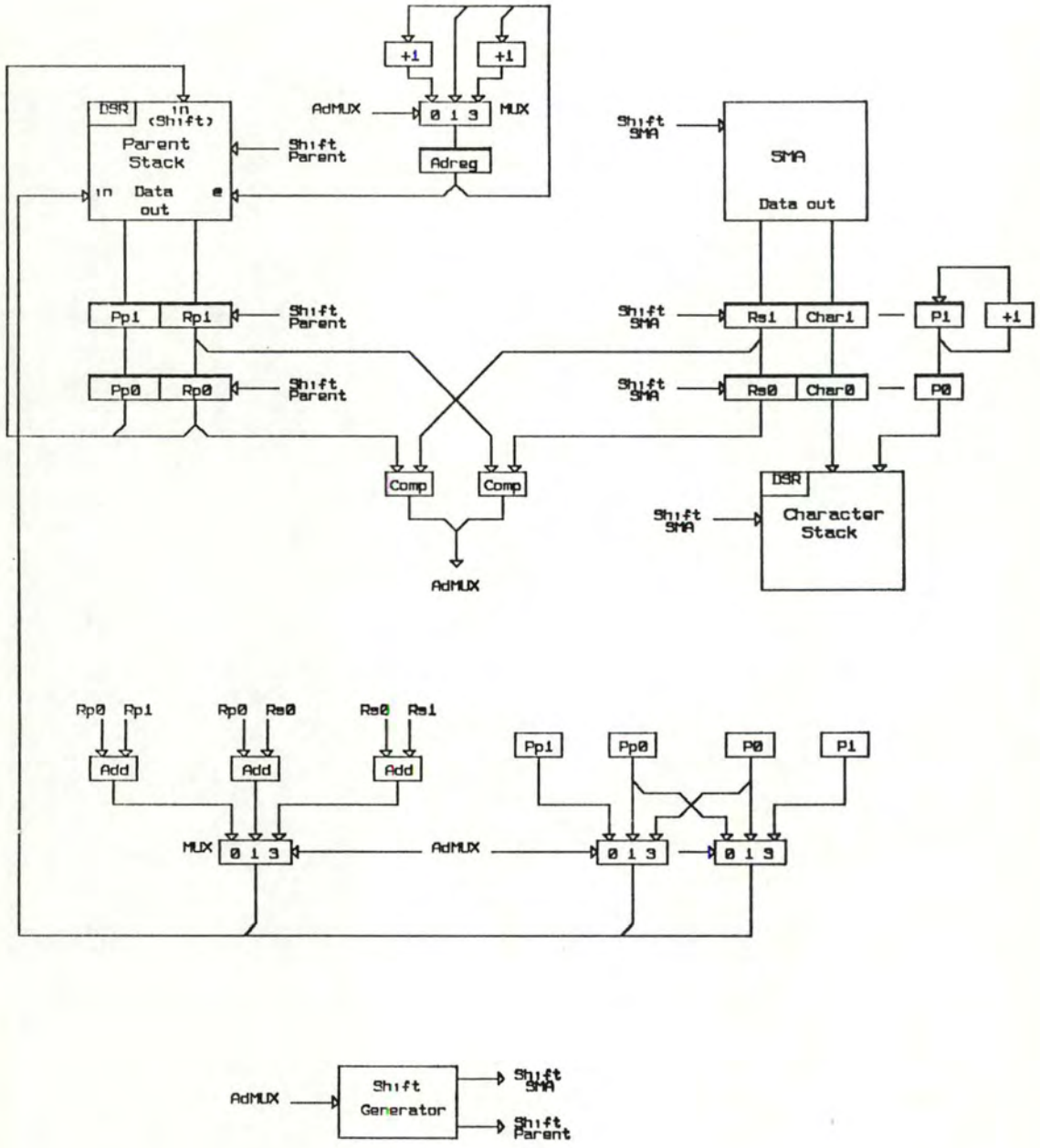
Its operation is carried out as follows :

A datum is presented at the "top" of the array with each clock period. As the smallest value in the array is always at the lower input of the Top comparator, there is no loss

of data, provided the array is large enough to accommodate the total numbers of characters.

The maximum sorting time, including entering the data, is twice the size of the array.

At the end of the sort, the contents of the array can be shifted out sequentially, starting from the highest or the lowest value as required.



DSR : Dynamic Stack/RAM

Add : Adder

Comp : Comparator

out	
1	A >= B
0	A < B

P0, P1, Pp0, Pp1 : Pointer registers

Rp0, Rp1, Rs0, Rs1 : Frequency registers

Figure 3 - Tree Constitution Logic

Step_3_Tree_Constitution

The following (figure 3) requires the design of a dedicated chip to control the data flow from the previously described Sorting Memory Array (SMA) to a "Parent Stack" and a "Character Stack". Note that these could be included in this chip.

Initially, the parent (dynamic) Stack-RAM is set high (I.E., all bits are logic 1) except the pointer fields, which are set to 256, 257, ...

At the start, 2 data are shifted out of the SMA, using the "Shift up" signal, and stored into Rs0 and Rs1. As Rp0 and Rp1 are both high, AdMUX will be 3, therefore constituting a "parent" with frequency (f0+f1), and pointers 0 and 1. AdMUX = 3 causes the SMA to be shifted twice, while the "parent" is stored in Rp0 (Adreg = 0 + 1), and the previous contents of Rs1 and Rs0's character fields are shifted in the character stack, together with their respective pointers (i.e. 1 and 0).

On the next operation, AdMUX can only be 1 or 3 ;

- In the first instance, the parent will have a frequency (f0+f1+f2), and pointers 2 and 256. AdMUX = 1 will cause the first parent to be shifted at the top of the "parent stack", while the new parent is stored in Rp0 (Adreg = 1 + 0). While this takes place, a new datum is shifted out of the SMA in Rs1, while Rs1 is shifted in Rs0, and Rs0's character field is shifted in the character stack with its pointer.
- In the second instance the parent will have a frequency (f2+f3), and pointers 2 and 3. The parent is then stored is Rp1 (Adreg = 1 + 1)

Let us now recapitulate for any operation :

AdMUX	parent frequency	parent pointers	Adreg	
0	f(pn)+f(p[n+1])	Pp[n+1],Ppn	Adreg-1	1)
1	f(pn)+f(n)	Pn,Ppn	Adreg	2)
3	f(n)+f(n+1)	Pn+1,Pn	Adreg+1	3)

1) - The "Parent Stack" is shifted twice.

- The new parent is stored while the second parent (in Rp1, initially), is shifted into the top of the parent stack.

- No shifts occur in the "Character Stack"/SMA groups.

2) - Both the "Parent Stack" and the "Character Stack"/SMA groups are shifted once.

- The new parent is stored during the shift.

3) - The "Character Stack"/SMA group is shifted twice.

- The new parent is stored during the shift.

In order to prepare for the next step, after the last parent has been constituted, signals derived from the state of Rp0, Rp1, Rs0 and Rs1 should be used to shift the "Parent Stack" until the total number of shifts that have occured in it equals 256. That way, the highest order node lies at location = total number of characters used - 1

Hardware implementation of various elements :

- The "dynamic Stack/RAM" (DSR) is a simple dynamic RAM with clocked transistor connections allowing the data to be shifted.
- Registers Rp0 and Rp1 could be part of this DSR, except that they both require parallel outputs.
- Any Adder, Incrementer, Decrementer, or Comparator may be easily constructed from basic switching networks.
- The various shift signal may be implemented by special feature 2 bit shift registers.
- Registers P0 and P1 are normal shift registers. P1 is incremented in between each shift.

Rp0 always holds the lowest unused (sum) frequency of the parent group, while Rs0 always holds the lowest unused frequency of the character group.

Notice that the two signal constituting "AdMUX" wholly describe in effect which registers contain the 2 lowest frequency values.

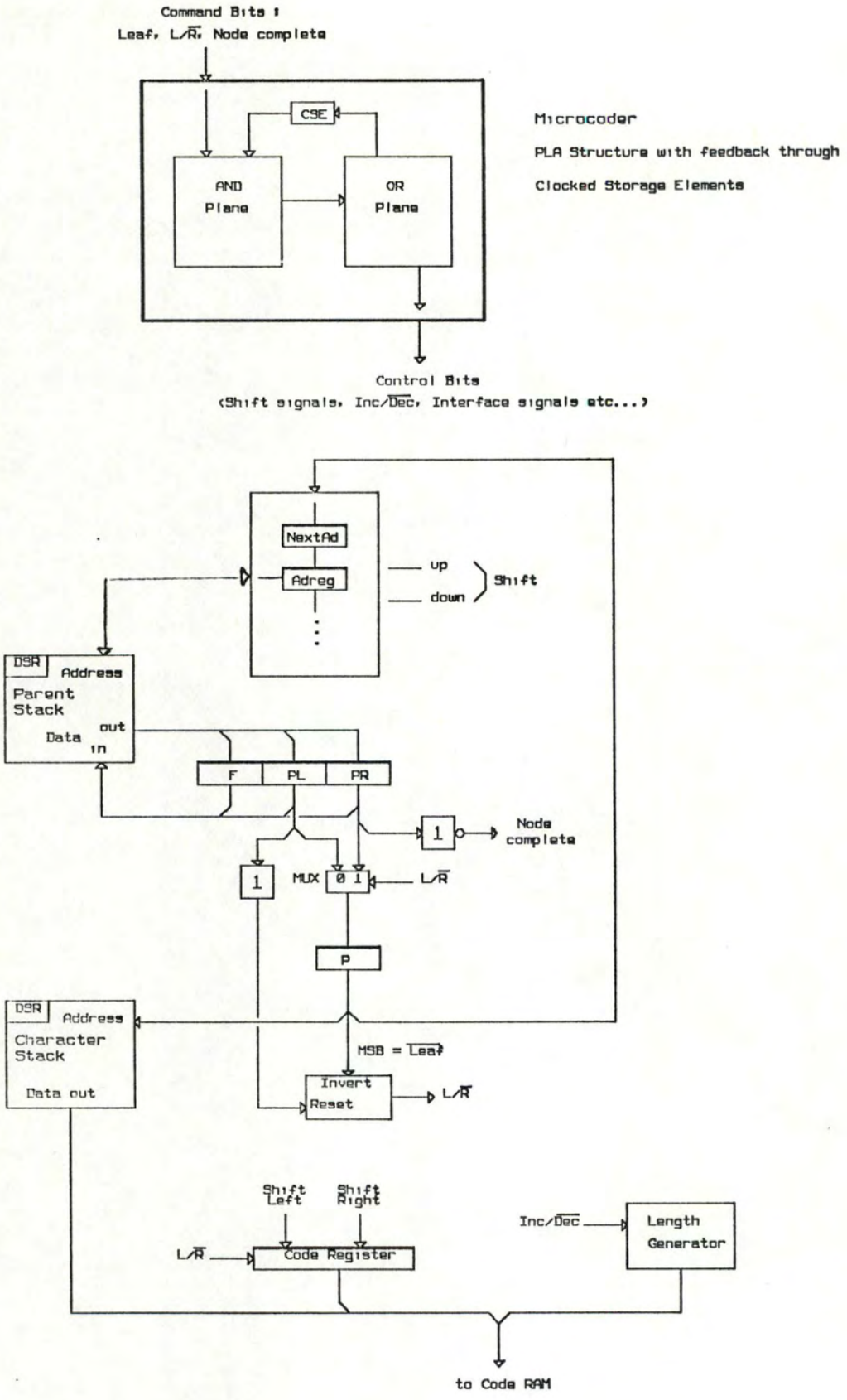


Figure 4 - Code Generator

Step_4_Coding

The main feature of this design approach is the use of a hardware implemented microcode control. This appears to be the more reasonable way, as the steps involved in going through each possible path of the tree in order to find its leaves can be a bit tricky.

The reader should constantly refer to figure 4 while following each step.

Initially, I/R is set to logic "0" (i.e. we start to constitute the tree from the left side) and the address output of the Address Stack Register (Adreg) is set to number of different characters - 1. Therefore, the data stored in TreeReg corresponds to the first node. The MSB of the selected pointer, namely P1 in this case, indicates whether we have reached a leaf. Supposing this is not the case, NextAdreg is loaded with P1, while a "0" is shifted in CodeReg, the length is incremented and the content of TreeReg is stored back in the Parent Stack. Then the Address Stack is shifted down so that the next data, corresponding to P1, can be fetched and loaded into TreeReg.

This operation carries on until a leaf is reached (MSB = 0). At this point, the following operations take place, as a character code can now be completed :

1) I/R is shifted in CodeReg and the length is incremented.

2) The character code, togetther with its length and the original code, is stored in the code RAM.

3) The length is decremented while CodeReg is shifted left (getting rid of the last entered bit) and the pointer field

corresponding to I/R is set to "0", indicating that this particular leaf has been found.

4) I/R is inverted while the content of TreeReg is stored back into the Parent Stack.

5) If the node is not completed, namely the content of Pr is not "0", Pr is stored in NextAdreg. If Pr indicates another leaf, steps 1) through 4) are repeated ; if not, the Address Stack is shifted down, and a new data is fetched.

If the node is completed, the Address Stack is shifted up while the length is incremented and CodeReg is shifted left. The parent node is then fetched, and step 5) is repeated

In general, when an "unexplored" node is reached, I/R is reset to "0" so that the step described initially can take place, the only difference being in the content of the address registers and stack upon completion of the coding, all pointer fields in the parent stack are "0".

Technology considerations

Although either bipolar or MOS technologies may be used, it appears that after a rapid estimate of the cost in silicon, preference should go to the latter. Some of the features are also more readily implemented in MOS ; though the cost in development may be higher than with bipolar gate arrays, for instance, a block such as the Sorting Memory Array would require several chips as gate array. Library cells tend to be pretty cumbersome for this kind of implementation. Also, a small hardware microcoder such as the one used in Step 4 would take very little room indeed in MOS.

Annexe 2 : Extrait de Base de Données PROLOG

Annexe 2.1 : Non codée

```
relat("city",["state","abbreviation","city","population"])
relat("river",["river","length","state"])
relat("point",["state","abbreviation","point","height"])
relat("border",["state","state"])
relat("mountain",["state","abbreviation","mountain","height"])
relat("lake",["lake","area","state"])
relat("road",["road","state"])
schema("abbreviation","of","state")
schema("state","with","abbreviation")
schema("capital","of","state")
schema("state","with","capital")
schema("population","of","state")
schema("state","with","population")
schema("area","of","state")
schema("city","in","state")
schema("state","with","city")
schema("population","of","city")
schema("population","of","capital")
schema("length","of","river")
schema("state","with","river")
schema("river","in","state")
schema("state","border","state")
schema("city","with","population")
schema("state","with","population")
schema("river","with","length")
schema("capital","with","population")
schema("point","in","state")
schema("state","with","point")
schema("height","of","point")
schema("mountain","in","state")
schema("state","with","mountain")
schema("height","of","mountain")
schema("lake","in","state")
schema("state","with","lake")
schema("area","of","lake")
schema("state","with","road")
schema("road","in","state")
schema("name","of","river")
schema("name","of","capital")
schema("name","of","city")
schema("name","of","state")
schema("name","of","point")
schema("name","of","mountain")
schema("name","of","lake")
schema("name","of","road")
schema("river","in","continent")
schema("city","in","continent")
schema("capital","in","continent")
schema("state","in","continent")
schema("point","in","continent")
schema("mountain","in","continent")
```

```
schema("lake","in","continent")
schema("road","in","continent")
entitysize("river","length")
entitysize("state","area")
entitysize("city","population")
entitysize("point","height")
entitysize("mountain","height")
entitysize("lake","area")
assoc("in",["in"])
assoc("in",["running","through"])
assoc("in",["runs","through"])
assoc("in",["run","through"])
assoc("with",["with"])
assoc("with",["traversed"])
assoc("with",["traversed","by"])
assoc("of",["of"])
assoc("border",["border"])
assoc("border",["frontier"])
assoc("border",["boundary"])
assoc("border",["limit"])
assoc("border",["borders"])
assoc("border",["border","on"])
assoc("of",["in"])
assoc("in",["of"])
assoc("with",["has"])
assoc("with",["have"])
synonym("people","population")
synonym("citizen","population")
synonym("inhabitant","population")
synonym("place","point")
synonym("town","city")
ignore("which")
ignore("is")
ignore("are")
ignore("the")
ignore("tell")
ignore("me")
ignore("what")
ignore("give")
ignore("as")
ignore("that")
ignore("please")
ignore("to")
ignore("how")
ignore("many")
ignore("live")
ignore("lives")
ignore("living")
ignore("there")
ignore("do")
ignore("does")
ignore("number")
ignore("a")
ignore("an")
ignore("any")
```

```
ignore("some")
city("alabama","al","birmingham",284413)
city("alabama","al","mobile",200452)
city("alabama","al","montgomery",177857)
city("alabama","al","huntsville",1425
city("alabama","al","tuscaloosa",75143)
city("alaska","ak","anchorage",174431)
city("arizona","az","phoenix",789704)
city("arizona","az","tucson",330537)
city("arizona","az","mesa",152453)
city("arizona","az","tempe",106919)
city("arizona","az","glendale",96988)
city("arizona","az","scottsdale",88622)
city("arkansas","ar","little rock",158915)
city("arkansas","ar","fort smith",71384)
city("arkansas","ar","north little rock",64388)
city("california","ca","los angeles",2966850)
city("california","ca","san diego",875538)
city("california","ca","san francisco",678974)
city("california","ca","san jose",629442)
city("california","ca","long beach",361334)
city("california","ca","oakland",339337)
city("california","ca","sacramento",275741)
city("california","ca","anaheim",219311)
city("california","ca","fresno",218202)
city("california","ca","santa ana",203713)
city("california","ca","riverside",170876)
city("california","ca","huntington beach",170505)
city("california","ca","stockton",149779)
city("california","ca","glendale",139060)
city("california","ca","fremont",131945)
city("california","ca","torrance",131497)
city("california","ca","garden grove",123351)
city("california","ca","san bernardino",118794)
city("california","ca","pasadena",118072)
city("california","ca","east los angeles",110017)
city("california","ca","oxnard",108195)
city("california","ca","modesto",106963)
city("california","ca","sunnyvale",106618)
city("california","ca","bakersfield",105611)
city("california","ca","concord",103763)
city("california","ca","berkeley",103328)
city("california","ca","fullerton",102246)
city("california","ca","inglewood",94162)
city("california","ca","hayward",93585)
city("california","ca","pomona",92742)
city("california","ca","orange",91450)
city("california","ca","ontario",88820)
city("california","ca","santa monica",88314)
city("california","ca","santa clara",87700)
city("california","ca","citrus heights",85911)
city("california","ca","norwalk",84901)
city("california","ca","burbank",84625)
city("california","ca","chula vista",83927)
city("california","ca","santa rosa",83205)
```

```
city("california","ca","downey",82602)
city("california","ca","costa mesa",82291)
city("california","ca","compton",81230)
city("california","ca","carson",81221)
city("california","ca","salinas",80479)
city("california","ca","west covina",80292)
city("california","ca","vallejo",80188)
city("california","ca","el monte",79494)
city("california","ca","daly city",78519)
city("california","ca","thousand oaks",77797)
city("california","ca","san mateo",77640)
city("california","ca","simi valley",77500)
city("california","ca","oceanside",76698)
city("california","ca","richmond",74676)
city("california","ca","lakewood",74654)
city("california","ca","santa barbara",74542)
city("california","ca","el cajon",73892)
city("california","ca","ventura",73774)
city("california","ca","westminster",71133)
city("california","ca","whittier",68558)
city("california","ca","south gate",66784)
city("california","ca","alhambra",64767)
city("california","ca","buena park",64165)
city("california","ca","san leandro",63952)
city("california","ca","alameda",63852)
city("california","ca","newport beach",63475)
city("california","ca","escondido",62480)
city("california","ca","irvine",62134)
city("california","ca","mountain view",58655)
city("california","ca","fairfield",58099)
city("california","ca","redondo beach",57102)
city("california","ca","scotts valley",6037)
city("colorado","co","denver",492365)
city("colorado","co","colorado springs",215150)
city("colorado","co","aurora",158588)
city("colorado","co","lakewood",113808)
city("colorado","co","pueblo",101686)
city("colorado","co","arvada",84576)
city("colorado","co","boulder",76685)
city("colorado","co","fort collins",64632)
city("connecticut","ct","bridgeport",142546)
city("connecticut","ct","hartford",136392)
city("connecticut","ct","new haven",126089)
city("connecticut","ct","waterbury",103266)
city("connecticut","ct","stamford",102466)
city("connecticut","ct","norwalk",77767)
city("connecticut","ct","new britain",73840)
city("connecticut","ct","west hartford",61301)
city("connecticut","ct","danbury",60470)
city("connecticut","ct","greenwich",59578)
city("connecticut","ct","bristol",57370)
city("connecticut","ct","meriden",57118)
city("delaware","de","wilmington",70195)
city("district of columbia","dc","washington",638333)
city("florida","fl","jacksonville",540920)
```

```
city("florida","fl","miami",346865)
city("florida","fl","tampa",271523)
city("florida","fl","st. petersburg",238647)
city("florida","fl","fort lauderdale",153256)
city("florida","fl","orlando",12
city("florida","fl","hollywood",117188)
city("florida","fl","miami beach",96298)
city("florida","fl","clearwater",85450)
city("florida","fl","tallahassee",81548)
city("florida","fl","gainesville",81371)
city("florida","fl","kendall",73758)
city("florida","fl","west palm beach",62530)
city("florida","fl","largo",58977)
city("florida","fl","pensacola",57619)
city("georgia","ga","atlanta",425022)
city("georgia","ga","columbus",169441)
city("georgia","ga","savannah",141654)
city("georgia","ga","macon",116860)
city("georgia","ga","albany",74425)
city("hawaii","hi","honolulu",762874)
city("hawaii","hi","ewa",190037)
city("hawaii","hi","koolaupoko",109373)
city("idaho","id","boise",102249)
city("illinois","il","chicago",3005172)
city("illinois","il","rockford",139712)
city("illinois","il","peoria",124160)
city("illinois","il","springfield",100054)
city("illinois","il","decatur",93939)
city("illinois","il","aurora",81293)
city("illinois","il","joliet",77956)
city("illinois","il","evanston",73706)
city("illinois","il","waukegan",67653)
city("illinois","il","arlington heights",66116)
city("illinois","il","elgin",63668)
city("illinois","il","cicero",61232)
city("illinois","il","oak lawn",60590)
city("illinois","il","skokie",60278)
city("illinois","il","champaign",58267)
city("indiana","in","indianapolis",700807)
city("indiana","in","fort wayne",172196)
city("indiana","in","gary",151968)
city("indiana","in","evansville",130496)
city("indiana","in","south bend",109727)
city("indiana","in","hammond",93714)
city("indiana","in","muncie",77216)
city("indiana","in","anderson",64695)
city("indiana","in","terre haute",61125)
city("iowa","ia","des moines",191003)
city("iowa","ia","cedar rapids",110243)
city("iowa","ia","davenport",103254)
city("iowa","ia","sioux city",82003)
city("iowa","ia","waterloo",75985)
city("iowa","ia","dubuque",62321)
city("kansas","ks","wichita",279212)
city("kansas","ks","kansas city",161148)
```

```
city("kansas","ks","topeka",118690)
city("kansas","ks","overland park",81784)
city("kentucky","ky","louisville",298451)
city("kentucky","ky","lexington",204165)
city("louisiana","la","new orleans",515675)
city("louisiana","la","baton rouge",219419)
city("louisiana","la","shreveport",205820)
city("louisiana","la","metairie",164160)
city("louisiana","la","lafayette",80584)
city("louisiana","la","lake charles",75051)
city("louisiana","la","kenner",66382)
city("louisiana","la","monroe",57597)
city("maine","me","portland",61572)
city("maryland","md","baltimore",786775)
city("maryland","md","silver spring",72893)
city("maryland","md","dundalk",71293)
city("maryland","md","bethesda",63022)
city("massachusetts","ma","boston",562994)
city("massachusetts","ma","worcester",161799)
city("massachusetts","ma","springfield",152319)
city("massachusetts","ma","new bedford",98478)
city("massachusetts","ma","cambridge",95322)
city("massachusetts","ma","brockton",95172)
city("massachusetts","ma","fall river",92574)
city("massachusetts","ma","lowell",92418)
city("massachusetts","ma","quincy",84743)
city("massachusetts","ma","newton",83622)
city("massachusetts","ma","lynn",78471)
city("massachusetts","ma","somerville",77372)
city("massachusetts","ma","framingham",65113)
city("massachusetts","ma","lawrence",63175)
city("massachusetts","ma","waltham",58200)
city("massachusetts","ma","medford",58076)
city("michigan","mi","detroit",1203339)
city("michigan","mi","grand rapids",181843)
city("michigan","mi","warren",161134)
city("michigan","mi","flint",159611)
city("michigan","mi","lansing",130414)
city("michigan","mi","sterling heights",108999)
city("michigan","mi","ann arbor",107969)
city("michigan","mi","livonia",104814)
city("michigan","mi","dearborn",90660)
city("michigan","mi","westland",84603)
city("michigan","mi","kalamazoo",79722)
city("michigan","mi","taylor",77568)
city("michigan","mi","saginaw",77508)
city("michigan","mi","pontiac",76715)
city("michigan","mi","st. clair shores",76210)
city("michigan","mi","southfield",75568)
city("michigan","mi","clinton",72400)
city("michigan","mi","royal oak",70893)
city("michigan","mi","dearborn heights",67706)
city("michigan","mi","troy",67102)
city("michigan","mi","waterford",64250)
city("michigan","mi","wyoming",59616)
```

```
city("michigan","mi","redford",58441)
city("michigan","mi","farmington hills",58056)
city("minnesota","mn","minneapolis",370951)
city("minnesota","mn","st. paul",270230)
city("minnesota","mn","duluth",92811)
city("minnesota","mn","bloomington",81831)
city("minnesota","mn","rochester",57906)
city("mississippi","ms","jackson",202895)
city("missouri","mo","st. louis",453085)
city("missouri","mo","kansas city",448159)
city("missouri","mo","springfield",133116)
city("missouri","mo","independence",111797)
city("missouri","mo","st. joseph",76691)
city("missouri","mo","columbia",62061)
city("montana","mt","billings",66842)
city("montana","mt","great falls",56725)
city("nebraska","ne","omaha",314255)
city("nebraska","ne","lincoln",171932)
city("nevada","nv","las vegas",164674)
city("nevada","nv","reno",100756)
city("new hampshire","nh","manchester",90936)
city("new hampshire","nh","nashua",67865)
city("new jersey","nj","newark",329248)
city("new jersey","nj","jersey city",223532)
city("new jersey","nj","paterson",137970)
city("new jersey","nj","elizabeth",106201)
city("new jersey","nj","trenton",92124)
city("new jersey","nj","woodbridge",90074)
city("new jersey","nj","camden",84910)
city("new jersey","nj","east orange",77878)
city("new jersey","nj","clifton",74388)
city("new jersey","nj","edison",70193)
city("new jersey","nj","cherry hill",68785)
city("new jersey","nj","bayonne",65047)
city("new jersey","nj","middletown",61615)
city("new jersey","nj","irvington",61493)
city("new mexico","nm","albuquerque",331767)
city("new york","ny","new york",7071639)
city("new york","ny","buffalo",357870)
city("new york","ny","rochester",241741)
city("new york","ny","yonkers",195351)
city("new york","ny","syracuse",170105)
city("new york","ny","albany",101727)
city("new york","ny","cheektowaga",92145)
city("new york","ny","utica",75632)
city("new york","ny","niagara falls",71384)
city("new york","ny","new rochelle",70794)
city("new york","ny","schenectady",67972)
city("new york","ny","mount vernon",66713)
city("new york","ny","irondequoit",57648)
city("new york","ny","levittown",57045)
city("north carolina","nc","charlotte",314447)
city("north carolina","nc","greensboro",155642)
city("north carolina","nc","raleigh",149771)
city("north carolina","nc","winston-salem",131885)
```

```
city("north carolina","nc","durham",100538)
city("north carolina","nc","high point",64107)
city("north carolina","nc","fayetteville",59507)
city("north dakota","nd","fargo",61308)
city("ohio","oh","cleveland",573822)
city("ohio","oh","columbus",564871)
city("ohio","oh","cincinnati",385457)
city("ohio","oh","toledo",354635)
city("ohio","oh","akron",237177)
city("ohio","oh","dayton",203371)
city("ohio","oh","youngstown",115436)
city("ohio","oh","canton",93077)
city("ohio","oh","parma",92548)
city("ohio","oh","lorain",75416)
city("ohio","oh","springfield",72563)
city("ohio","oh","hamilton",63189)
city("ohio","oh","lakewood",61963)
city("ohio","oh","kettering",61186)
city("ohio","oh","euclid",59999)
city("ohio","oh","elyria",57504)
city("oklahoma","ok","oklahoma city",403213)
city("oklahoma","ok","tulsa",360919)
city("oklahoma","ok","lawton",80054)
city("oklahoma","ok","norman",68020)
city("oregon","or","portland",366383)
city("oregon","or","eugene",105664)
city("oregon","or","salem",89233)
city("pennsylvania","pa","philadelphia",1688210)
city("pennsylvania","pa","pittsburgh",423938)
city("pennsylvania","pa","erie",119123)
city("pennsylvania","pa","allentown",103758)
city("pennsylvania","pa","scranton",88117)
city("pennsylvania","pa","upper darby",84054)
city("pennsylvania","pa","reading",78686)
city("pennsylvania","pa","bethlehem",70419)
city("pennsylvania","pa","lower merion",59651)
city("pennsylvania","pa","abingdon",59084)
city("pennsylvania","pa","bristol township",58733)
city("pennsylvania","pa","penn hills",57632)
city("pennsylvania","pa","altoona",57078)
city("rhode island","ri","providence",156804)
city("rhode island","ri","warwick",87123)
city("rhode island","ri","cranston",71992)
city("rhode island","ri","pawtucket",71204)
city("south carolina","sc","columbia",101229)
city("south carolina","sc","charleston",69855)
city("south carolina","sc","north charleston",62504)
city("south carolina","sc","greenville",58242)
city("south dakota","sd","sioux falls",81343)
city("tennessee","tn","memphis",646356)
city("tennessee","tn","nashville",455651)
city("tennessee","tn","knoxville",175030)
city("tennessee","tn","chattanooga",169728)
city("texas","tx","houston",1595138)
city("texas","tx","dallas",904078)
```

```
city("texas","tx","san antonio",785880)
city("texas","tx","el paso",425259)
city("texas","tx","fort worth",385164)
city("texas","tx","austin",345496)
city("texas","tx","corpus christi",231999)
city("texas","tx","lubbock",173979)
city("texas","tx","arlington",160123)
city("texas","tx","amarillo",149230)
city("texas","tx","garland",138857)
city("texas","tx","beaumont",118102)
city("texas","tx","pasadena",112560)
city("texas","tx","irving",109943)
city("texas","tx","waco",101261)
city("texas","tx","abilene",98315)
city("texas","tx","wichita falls",94201)
city("texas","tx","laredo",91449)
city("texas","tx","odessa",90027)
city("texas","tx","brownsville",84997)
city("texas","tx","san angelo",73240)
city("texas","tx","richardson",72496)
city("texas","tx","plano",72331)
city("texas","tx","grand prairie",71462)
city("texas","tx","midland",70525)
city("texas","tx","tyler",70508)
city("texas","tx","mesquite",67053)
city("texas","tx","mcallen",67042)
city("texas","tx","longview",62762)
city("texas","tx","port arthur",61195)
city("utah","ut","salt lake city",163034)
city("utah","ut","provo",74111)
city("utah","ut","west valley",72299)
city("utah","ut","ogden",64407)
city("virginia","va","norfolk",266979)
city("virginia","va","virginia beach",262199)
city("virginia","va","richmond",219214)
city("virginia","va","arlington",152599)
city("virginia","va","newport news",144903)
city("virginia","va","hampton",122617)
city("virginia","va","chesapeake",114226)
city("virginia","va","portsmouth",104577)
city("virginia","va","alexandria",103217)
city("virginia","va","roanoke",100427)
city("virginia","va","lynchburg",66743)
city("washington","wa","seattle",493846)
city("washington","wa","spokane",171300)
city("washington","wa","tacoma",158501)
city("washington","wa","bellevue",73903)
city("west virginia","wv","charleston",63968)
city("west virginia","wv","huntington",63684)
city("wisconsin","wi","milwaukee",636212)
city("wisconsin","wi","madison",170616)
city("wisconsin","wi","green bay",87899)
city("wisconsin","wi","racine",85725)
city("wisconsin","wi","kenosha",77685)
city("wisconsin","wi","west allis",63982)
```

Annexe_5.2_:_Après_Codage_par_LOVER

```
relat("city",["state","abbreviation","city","population"])
æææææææræver",["rivær","length","state"])
æææææææpointæææææstateæææabbreviation",æpoint","height"])
æææææææhærder",["state","state"])
ææææææææmæuntain",["sæateæ,"æbbreviation","mountain","height"])
ææææææælake",["lake","areaæ,"state"])
æææææææroadæææærroadæææstate"])
schemæ("abbreviation","of",æstate")
æææææææææstatæ","wæthæææabbreviææion")
æææææææææcapitalæ,"ofæææstatæ")
æææææææææstate",æwithæææcapital")
æææææææææpopulation",æof","sææte")
æææææææææstate","withæ,"populææion")
æææææææææarea","of","state")
æææææææææcityæææinæææææææææææ
æææææææææstate","with","city")
æææææææpopulation",æof","city")
æææææææææææææææææææææææææææææapital")
æææææææææength","of","river")
æææææææææstate","with","river")
æææææææææriverææææin",æstate")
æææææææææstateæææborder","state")
æææææææææcity","with","populææion")
æææææææææstate","with","population")
æææææææææriverææææææææææength")
æææææææææcapitalæ,"withæ,"population")
æææææææææpoinæ",æin","state")
æææææææææstateæææwithæ,"point")
æææææææææheight","of","point")
æææææææææmountainæ,"inæ,"state")
æææææææææstate","with","mounæain")
æææææææææheight","of","mountain")
ææææææææælake",æin",æstate")
æææææææææstate","with","lake")
æææææææææarea","of","lake")
æææææææææstate","with","road")
æææææææææroæd","in","state")
ææææææææænameæææofæææriverææ
æææææææææææææææææææææææcapital")
æææææææææææææææææææææææity")
æææææææææææææææææææææstate")
æææææææææææææææææææpointææ
æææææææææææææææææææææmæuæææain")
ææææææææææææææææææææælake")
æææææææææææææææææææroadææ
æææææææææriværæ","in","continent")
æææææææææcæty","in","continent")
æææææææææapital",æin","coætinent")
æææææææææstate",æin",æcontineæt")
æææææææææpointæææææææææææææææææ
æææææææææmæuæææain",æin","coætinent")
ææææææææælake","in","continent")
æææææææææroadæææææææææææææææææ
```

```
entitysize("riverææælEægth")
æææææææææææææstateæææarea")
æææææææææææææcity","population")
æææææææææææææpoint","heighæ")
æææææææææææææmæuææain","height")
ææææææææææææælake","area")
assoc("æn",["in"])
æææææææææææææærunning","through"])
ææææææææææææææææææs","through"])
æææææææææææææææææææ","through"])
æææææææwith",["with"])
ææææææææææææææææætraversed"])
æææææææææææææææææææææææææææææææ,"by"])
æææææææof",["of"])
æææææææborder",["border"])
æææææææææææææææææææfrontier"])
æææææææææææææææææææbouædaryæææ
æææææææææææææææææææl imit"])
æææææææææææææææææææborders"])
ææææææææææææææææææææææææææææ","on"])
ææææææææof",["in"])
æææææææinæææofæææ
ææææææææwith",["has"])
ææææææææææææææææææææve"])
synænym("people","population")
ææææææææææcitizæn","population")
æææææææææinhabitant","population")
æææææææææplace","point")
æææææææææætown","city")
igæære("whichæ)
ææææææææææis")
ææææææææææare")
ææææææææææthæææ
æææææææææææell")
æææææææææææmæ")
ææææææææææwhat")
ææææææææææ give ææ
ææææææææææas")
ææææææææææthat")
ææææææææææplease")
ææææææææææto")
ææææææææææhæw")
ææææææææææmany")
ææææææææææliveææ
æææææææææææææææs")
æææææææææææææing")
ææææææææææthere")
ææææææææææ do")
ææææææææææææes")
æææææææææænumber")
ææææææææææa")
æææææææææææn")
æææææææææææææy")
ææææææææææsome")
city("alabama","al","birmingham",284413)
```

```
^[æææææææææææææææææææææææææææææmobile",200452)
æææææææææææææææææææææææææææææææntgomery",177857)
æææææææææææææææææææææææææææææhuææsvilleæææ425
ææææææææææææææææææææææææææææætæscaloosaææ75143)
ææææææææææææska","ak","anchorage",174431)
ææææææææerizona","az","phoenix",789704)
æææææææææææææææææææææææææææææætucson",330537)
æææææææææææææææææææææææææææææmesa",152453)
æææææææææææææææææææææææææææææætæmpe",106919)
æææææææææææææææææææææææææææææglendale",æ6æ88)
æææææææææææææææææææææææææææææscottsdale",8æ622)
æææææææææææææækansæs","ar","liææle rock",158915)
æææææææææææææææææææææææææææææforæ smith",7æ3æ4)
ææææææææææææææææææææææææææææææænæææh lææle rock",64388)
ææææææææecaliforniaæ,"ææ","losæangeææs",2966850)
æææææææææææææææææææææææææææææææsanædiego",875538)
ææææææææææææææææææææææææææææææææææfrancisco",678974)
ææææææææææææææææææææææææææææææææjose",629442)
ææææææææææææææææææææææææææææææææloæg beach",361334)
ææææææææææææææææææææææææææææææææoakland",339æ37)
ææææææææææææææææææææææææææææææææsæcræmento",275741)
ææææææææææææææææææææææææææææææææanaheim",219311)
ææææææææææææææææææææææææææææææfresno",218202)
ææææææææææææææææææææææææææææææææsanta ana",ææ3713)
æææææææææææææææææææææææææææææææriversideææ170876æ
ææææææææææææææææææææææææææææææææhuntington beach",170505)
ææææææææææææææææææææææææææææææææstockton",149779)
æææææææææææææææææææææææææææææææglendaleæææ3æ060æ
æææææææææææææææææææææææææææææææfræmont",131æ45)
æææææææææææææææææææææææææææææætorraæce",131æ97)
ææææææææææææææææææææææææææææææææaædeæ grove",123351)
ææææææææææææææææææææææææææææææææsæn bernardino",118794)
ææææææææææææææææææææææææææææææææpæsadæna",118072)
ææææææææææææææææææææææææææææææææeææt los angeles",110017)
æææææææææææææææææææææææææææææææoxnard",108195)
æææææææææææææææææææææææææææææææmodesto",106æ63)
ææææææææææææææææææææææææææææææsunnyvale",10æ618)
ææææææææææææææææææææææææææææææbakersfiæld",105611)
æææææææææææææææææææææææææææææææconcord",103763)
æææææææææææææææææææææææææææææææberkeley",103328)
ææææææææææææææææææææææææææææææææfullærton",102æ46)
ææææææææææææææææææææææææææææææææingææwoædææ94162)
ææææææææææææææææææææææææææææææhayward",93585)
ææææææææææææææææææææææææææææææpomona",92742)
ææææææææææææææææææææææææææææææææorangeææææ1450æ
æææææææææææææææææææææææææææææææææntario",88820)
ææææææææææææææææææææææææææææææææsanta monica",88314)
ææææææææææææææææææææææææææææææææææææææclara",87700)
ææææææææææææææææææææææææææææææææcitrus heights",85911)
æææææææææææææææææææææææææææææænorwalk",84901)
ææææææææææææææææææææææææææææææbuæbænææææ625æ
ææææææææææææææææææææææææææææææchulæ vista",83927)
ææææææææææææææææææææææææææææææææsantææroæa",83205)
ææææææææææææææææææææææææææææææææædowney",82602)
ææææææææææææææææææææææææææææææææcæsta mesa",82291)
```

"[ææææææææææææææææææææææææææææææææmpton",81230)
æææææææææææææææææææææææææææææææarson",812æ1)
ææææææææææææææææææææææææææææææsælinas",80479)
ææææææææææææææææææææææææææææwest covina",80292)
æææææææææææææææææææææææææææææævallejæ",8018æ)
æææææææææææææææææææææææææææææææel monte",79494)
æææææææææææææææææææææææææææææææææææææædaly city",78519)
æææææææææææææææææææææææææææææthousand oaks",77797)
ææææææææææææææææææææææææææææææsan mæteo",77640)
ææææææææææææææææææææææææææææææimi valley",77500)
ææææææææææææææææææææææææææææææææoceanside",76698)
ææææææææææææææææææææææææææææææricmonæ",74æ76)
ææææææææææææææææææææææææææææææælakewæoæææææ54æ
ææææææææææææææææææææææææææææææsænta barbara",74542)
æææææææææææææææææææææææææææææel cæjon",73892)
ææææææææææææææææææææææææææææææventura",73774)
ææææææææææææææææææææææææææææææwæsæminster",71133)
æææææææææææææææææææææææææææææhiætæer",68558)
ææææææææææææææææææææææææææææsouæh gate",66784)
ææææææææææææææææææææææææææææææalhambræ",647ææ)
ææææææææææææææææææææææææææææbuena pærk",64165)
ææææææææææææææææææææææææææææsan leandro",639æ2)
ææææææææææææææææææææææææææææææalamedæ",63852)
ææææææææææææææææææææææææææææænewport beach",63475)
æææææææææææææææææææææææææææææescondido",62480)
ææææææææææææææææææææææææææææææirviæe",62134)
æææææææææææææææææææææææææææææmountain view",58655)
æææææææææææææææææææææææææææææfairfield",58099)
æææææææææææææææææææææææææææredondo beach",57102)
æææææææææææææææææææææææææææææscotts valleyææ6037)
ææææææææoæorado",æco",ædenver",492365)
ææææææææææææææææææææææææcolorado springs",215150)
ææææææææææææææææææææææææææææææauræææ",158588)
ææææææææææææææææææææææææææææælakewood",113æ08)
æææææææææææææææææææææææææææææpueblæ",10æ686)
æææææææææææææææææææææææææææææææarvadaææ84576)
ææææææææææææææææææææææææææææboulæer",76685)
æææææææææææææææææææææææææææfært collins",64632)
ææææææææææmnecticut",æct","bridgeport",142546)
ææææææææææææææææææææææææææææææhartford",136392)
æææææææææææææææææææææææææææææænew haven",126089)
ææææææææææææææææææææææææææææææææwaterburyæææ03266æ
ææææææææææææææææææææææææææææææstamford",10246æ)
ææææææææææææææææææææææææææææææænorwalk",77767)
ææææææææææææææææææææææææææææææææew britain",æ3840)
ææææææææææææææææææææææææææææææwæst hartford",61301)
æææææææææææææææææææææææææææææædanbury",60470)
ææææææææææææææææææææææææææææææægreenwich",59578)
ææææææææææææææææææææææææææææææbæistol",57370)
ææææææææææææææææææææææææææææææmeridenæææææ118æ
æææææææædelaware","deæ,"wilæingtoæææ70æ95æ
æææææææistrict of columbia","dc","washington",638333)
ææææææflorida","æl","jacksonvilleæ,540920)
ææææææææææææææææææææææææææææmiami",346865)
æææææææææææææææææææææææææætampaææ271523æ

```
^[æææææææææææææææææææææst. petersburg",238647)
æææææææææææææææææææææfort lauderdale",153256)
æææææææææææææææææææææorlando",12
æææææææææææææææææææææhoælywæod",117188)
æææææææææææææææææææææmiami beach",96298)
æææææææææææææææææææææclearwater",85450)
ææææææææææææææææææææætallahæssee",81æ48)
ææææææææææææææææææææægæinesvillæææææ371æ
ææææææææææææææææææææækendall",7375æ)
æææææææææææææææææææææwæst palm beach",62530)
ææææææææææææææææææææælargo",58977)
æææææææææææææææææææææpensacola",57619)
æææææægæægiæææægaæææatlanta",42æ022)
æææææææææææææææææææcoæumbus",169441)
æææææææææææææææææææsavannahæææ41654æ
æææææææææææææææææææmæcoæ",116860)
æææææææææææææææææææalbaæy",74425æ
æææææhawaiæ","hi","honolulu",76æ874)
æææææææææææææææææææewa",19003æ)
ææææææææææææææææækoolaupoko",109373)
æææææidaho","id","boise",102249)
ææææællinois",æil","chicago",3005172)
ææææææææææææææææææærockford",13971ææ
æææææææææææææææææpeoria",124160)
æææææææææææææææææspringfield",100054)
ææææææææææææææææædecatur",93939)
æææææææææææææææææaurora",812ææ)
æææææææææææææææjolietææ77956æ
æææææææææææææææææevansæon",73706)
æææææææææææææææwaukegaæææ67653æ
æææææææææææææææarlinæton heights",66116)
æææææææææææææææelgææ",63668)
æææææææææææææicero",61232)
æææææææææææææoak lawn",60590)
æææææææææææææææskokie",60278)
æææææææææææææchampaign",5æ267)
ææææændæona","in","indianæpolis",700807)
æææææææææææfort wæyne",172196)
ææææææææææææææææægaæy",151968)
æææææææææææææevansville",130496)
æææææææææææææsouth bendæææ09727æ
æææææææææææææhammond",937æ4)
æææææææææææææmuncie",77216)
æææææææææææææanderson",64æ95)
æææææææææææææterre haute",61125)
æææææwa","iaæ,"des moines",191003)
æææææææææææææcædar rapids",æ1æ243)
ææææææææææææææædavenport",103254)
æææææææææææææsioux ciæy",82003æ
æææææææææææææææwaterloo",75985)
ææææææææææææææædubuque",62321)
ææææækansasæ,"ksæ,"wichita",279212)
ææææææææææææææækansas city",æ61148)
ææææææææææææææææætopeka",118690)
æææææææææææææææææoverlænd park",81784)
```

^[ææææææætuckyæ,"kyæ,"louisville",298451æ
æææææææææææææææææææææææææææææexington",20416æ)
æææææælouisiana","la","new orleans",51æ675)
æææææææææææææææææææææææææææbaton rougeææ2æ9419æ
æææææææææææææææææææææææææææshrevepært",205820)
æææææææææææææææææææææææææææmetairie",164160)
ææææææææææææææææææææææææææælafæyette",80584æ
ææææææææææææææææææææææææææææke charles",75051)
æææææææææææææææææææææææææææækenner",66382)
æææææææææææææææææææææææææææmoæroeææ57597æ
ææææææmaine","me","portlæd",61ææ2)
æææææææææryland",æmd","baltimore",786775)
æææææææææææææææææææææææææææsiæver spring",æ2893)
ææææææææææææææææææææææææææælundalk",71293)
æææææææææææææææææææææææææbethesda",63022)
ææææææææææsachusettsæææma","boston",562994)
ææææææææææææææææææææææææææææææwærcester",161799)
æææææææææææææææææææææææææææææææspæingfield",æ52319)
æææææææææææææææææææææææææææææææænew bedforææ98478)
ææææææææææææææææææææææææææææææææcambriæge",95322)
æææææææææææææææææææææææææææææææbrockton",9517æ)
ææææææææææææææææææææææææææææææææfall river",92574)
æææææææææææææææææææææææææææææææælowell",92418)
æææææææææææææææææææææææææææææææquincyææ84743æ
æææææææææææææææææææææææææææææææænewtonæææ3622æ
ææææææææææææææææææææææææææææææææælynn",78471)
ææææææææææææææææææææææææææææææææsomerville",77372)
ææææææææææææææææææææææææææææææææframinghamææ65113æ
ææææææææææææææææææææææææææææææææælawreæce",63175)
æææææææææææææææææææææææææææææææwæltham",58200)
æææææææææææææææææææææææææææææææææmedfordææææ076æ
æææææææichigan","miææædetroit",1203339)
ææææææææææææææææææææææææææææægrand rapids",181843)
ææææææææææææææææææææææææææææwarren",161134)
æææææææææææææææææææææææææææflint",1596æ1)
ææææææææææææææææææææææææææææælansing",130414)
ææææææææææææææææææææææææææææsterling heights",108999)
ææææææææææææææææææææææææææææann arbor",107969)
æææææææææææææææææææææææææææælivonia",104814)
ææææææææææææææææææææææææææææædearborn",90660æ
ææææææææææææææææææææææææææææwæstlandææ84æ03æ
ææææææææææææææææææææææææææææææækalamæzoo",79722)
ææææææææææææææææææææææææææææætæylor",77568)
æææææææææææææææææææææææææææææsægingaw",æ7508)
æææææææææææææææææææææææææææææpontiæcææ6715æ
ææææææææææææææææææææææææææææst. clair shores",76210)
æææææææææææææææææææææææææææææouthfield",75568)
ææææææææææææææææææææææææææææclinton",72400)
æææææææææææææææææææææææææææææroyal oak",7æ893)
ææææææææææææææææææææææææææææædearborn heights",67706)
æææææææææææææææææææææææææææætroy",67102)
æææææææææææææææææææææææææææææwaterford",64250)
æææææææææææææææææææææææææææææyoming",59æ16)
æææææææææææææææææææææææææææææredfordæææ8441æ
æææææææææææææææææææææææææææfarmington hills",58056)

```
^[ææææææææænnesota","mn","minneapælis",3709æ1)
ææææææææææææææææææææææææææææææææææææst. pæul",27023æ)
æææææææææææææææææææææææææææææææææææææææduluth",92811)
ææææææææææææææææææææææææææææææææææææææææbloomington",81831)
æææææææææææææææææææææææææææææææææææææææ rochester",57906)
ææææææææææææææææssiæsippiæ,"msæ,"jacksonææ202895)
æææææææææææææouræ","moæ,"st. louis",4530æ5)
æææææææææææææææææææææææææææææææææææææækansas city",44æ159)
ææææææææææææææææææææææææææææææææææææspringfieldææ133æ16æ
æææææææææææææææææææææææææææææææææææææindependænce",11æ797)
æææææææææææææææææææææææææææææææææææst. joseph",7669æ)
æææææææææææææææææææææææææææææææææcolumbia",620æ1)
æææææææææææntana","mt","bilæings",6æ842)
ææææææææææææææææææææææææææææææææægreat falls",56725)
æææææææææebræska","ne","omæha",31425æ)
æææææææææææææææææææææææææææææææææælincoln",171932)
æææææææææævada",ænv",ælas vegasæææ64674æ
ææææææææææææææææææææææææææææææææreno",100756)
ææææææææææææw hæmpshireæ,"æbææ"manchester",90936)
æææææææææææææææææææææææææææææææææææææææææ næshua",67865)
æææææææææææææææjersey","njæ,"newark",32924æ)
æææææææææææææææææææææææææææææææææææææjærsey city",223532)
æææææææææææææææææææææææææææææææææææææpaterson",137970)
æææææææææææææææææææææææææææææææææææææelizabeth",1062æ1)
æææææææææææææææææææææææææææææææææææææætrenton",92æ24)
æææææææææææææææææææææææææææææææææææææwoodbridge",90074)
æææææææææææææææææææææææææææææææææææææcamæen",84910)
æææææææææææææææææææææææææææææææææææææeæst orange",77878)
ææææææææææææææææææææææææææææææææææææææcliftæn",74388)
ææææææææææææææææææææææææææææææææææææææææedæson",70193)
æææææææææææææææææææææææææææææææææææææcherry hill",68785)
ææææææææææææææææææææææææææææææææææææbayonne",65047)
ææææææææææææææææææææææææææææææææææææmiddletown",61615)
ææææææææææææææææææææææææææææææææææææææirvingæn",61493)
æææææææææææææææmæxicoæææmæææalbuquerque",331767)
ææææææææææææææyork",æny",ænew york",70716æ9)
ææææææææææææææææææææææææææææææææææææbuffalo",357870)
æææææææææææææææææææææææææææææææææææææ rochester",241741)
ææææææææææææææææææææææææææææææææææææyænkærs",195351)
æææææææææææææææææææææææææææææææææææææsyracuæe",170105)
ææææææææææææææææææææææææææææææææææalbæny",10ææ27)
ææææææææææææææææææææææææææææææææææææcheektowaga",92145)
ææææææææææææææææææææææææææææææææææææutica",75632)
æææææææææææææææææææææææææææææææææææææniagæra falls",71384)
ææææææææææææææææææææææææææææææææææææew rochelæe",70794)
ææææææææææææææææææææææææææææææææææææschenecætady",6æ9æ2)
æææææææææææææææææææææææææææææææææææmount vernon",66æ13)
æææææææææææææææææææææææææææææææææææiroædequoit",57æ48)
æææææææææææææææææææææææææææææææææææ levittown",57045)
ææææææææææorth carolina","nc","charlotte",314447)
ææææææææææææææææææææææææææææææææææææææææææææ greensboro",æ55642)
æææææææææææææææææææææææææææææææææææææææææææ ralæigh",149771)
ææææææææææææææææææææææææææææææææææææææææææææwinston-salem",131885)
æææææææææææææææææææææææææææææææææææææææææææ durham",100538)
æææææææææææææææææææææææææææææææææææææææææææhigæ point",64107)
```

```
^[ææææææææææææææææææææææææææææææfayetteville",59507)
æææææææææææææædækæta",æud",æfargo",61308)
æææææohio","oh","cleveland",573822)
æææææææææææææææææææææææolumbus",564871)
æææææææææææææææææææææææincinnati",3æ5457)
ææææææææææææææææææææææætoledo",3546æ5)
ææææææææææææææææææææææakron",2æ7177)
æææææææææææææææææææææædayton",2033æ1)
æææææææææææææææææææææææyoungstown",1æ5436)
ææææææææææææææææææææææcanton",93077)
ææææææææææææææææææææææpærma",92548)
æææææææææææææææææææææælaæain",7ææ16)
ææææææææææææææææææææææspæingfield",72563)
æææææææææææææææææææææææhammlton",63189)
ææææææææææææææææææææææælækewoædæææ1963æ
æææææææææææææææææææææækettering",61186)
ææææææææææææææææææææææeuclid",59999)
æææææææææææææææææææææææææælyrææææ7504æ
ææææææklahoma","ok","oklahoma city",403213)
æææææææææææææææææææææææææætuæsa",360919)
ææææææææææææææææææææææææææælawton",8æ054æ
ææææææææææææææææææææææææææænormaæææ68æ20æ
æææææææregon",æor",æportlandææ366383)
æææææææææææææææææææææææææeugene",105ææ4)
ææææææææææææææææææææææææææsalæm",89233)
ææææææpennsylvania","pa",æphiladelphia",1688210)
æææææææææææææææææææææææææææææææittsburgæ",42393æ)
ææææææææææææææææææææææææææææææææerie",119123)
ææææææææææææææææææææææææææææææææallæntown",103758)
æææææææææææææææææææææææææææææææscraæææn",8811æ)
æææææææææææææææææææææææææææææææupper darby",84054)
æææææææææææææææææææææææææææææææreading",78686)
ææææææææææææææææææææææææææææææææbæthlehem",70419)
ææææææææææææææææææææææææææææææææælower mærion",5æ651)
æææææææææææææææææææææææææææææææabingdon",59084)
æææææææææææææææææææææææææææææææææbræstol township",58733)
ææææææææææææææææææææææææææææææææpenn hills",57632)
æææææææææææææææææææææææææææææææææaltoona",57078)
ææææææærhode islandæææriææprovidence",156804)
æææææææææææææææææææææææææææææææææwarwæck",87123)
æææææææææææææææææææææææææææææææææcranston",ææ992)
ææææææææææææææææææææææææææææææææææpawtuckæt",71204)
ææææææsouthæcarolinaæ,"scæ,"columbia",æ01229)
ææææææææææææææææææææææææææææææææææharleston",69855)
æææææææææææææææææææææææææææææææææænorth charleston",62504)
æææææææææææææææææææææææææææææææææægreenville",58242)
ææææææææææææææædækæta",æsd",æsioux falæs",81343)
æææææætennessee","tnæ,"memphis",646356)
æææææææææææææææææææææææææææænashvælle",45ææ51)
ææææææææææææææææææææææææææææknoxæææææææ17æ030æ
æææææææææææææææææææææææææææææchattanooga",169728)
ææææææææexas","tx","houston",1595138)
æææææææææææææææææææææææææædallas",904078)
æææææææææææææææææææææææææææsæn æntonio",785880)
æææææææææææææææææææææææææel pæso",425259)
```

```
^[æææææææææææææææææææfort wærth",385164)
æææææææææææææææææææaustin",345496)
æææææææææææææææææcorpus christi",231999)
æææææææææææææææææææubbock",173979)
æææææææææææææææææææarlington",160123)
æææææææææææææææææmarillæ",149230)
ææææææææææææææææægarland",138857)
æææææææææææææææææbeaumont",11æ102)
æææææææææææææææææpasadeæaæææ2560æ
æææææææææææææææææirving",109943)
æææææææææææææææææwaco",10æ261)
æææææææææææææææææabilene",98315)
æææææææææææææææææwichita falls",94201)
æææææææææææææææææaredo",91449)
æææææææææææææææææodessaæææ0027æ
æææææææææææææææææbrownsville",84997)
æææææææææææææææææsan angeæo",73240)
æææææææææææææææææichærdsonæææ2496æ
æææææææææææææææææplano",72331)
æææææææææææææææææægræed prairie",71462)
æææææææææææææææææmidland",70525)
æææææææææææææææææyler",705æ8)
æææææææææææææææææmesquite",67053)
æææææææææææææææææallen",67042)
æææææææææææææææææongview",62762)
æææææææææææææææææpert arthur",æ1195)
æææææææutah","uæ","salt læke city",æ63034)
æææææææææææææææææprovo",74111)
æææææææææææææææææest valley",72299)
æææææææææææææææææogden",64407)
æææææææirginia","va","æorfolk",266979)
æææææææææææææææææirginia beach",262199)
æææææææææææææææææichmond",219214)
æææææææææææææææææarlington",æ5æ599)
æææææææææææææææææewporæ æews",144903)
æææææææææææææææææhamæton",122617)
æææææææææææææææææhesapeake",1æ4226)
æææææææææææææææææortsmouthæææ0æ577æ
æææææææææææææææææalexandriaæææ321ææ
æææææææææææææææææoanoke",100427)
æææææææææææææææææynchburg",66743)
æææææææashingtonæ,"waæ,"seattleæ493846)
æææææææææææææææææpokaneæææ171300æ
æææææææææææææææææacoma",158501)
æææææææææææææææææellevue",73903)
æææææææt virginia",æwv","charleston",63968)
æææææææææææææææææhuntingæææææææ684æ
æææææææisconsin","wi","milwaækee",636212)
æææææææææææææææææadison",170616)
æææææææææææææææææreen bay",87899)
æææææææææææææææææracine",85725)
æææææææææææææææææenosha",7æ685)
æææææææææææææææææwæst allis",63982)
```

Source : Turbo Prolog (Borland Int. Inc.) - Geobase.dba (Extrait)

Annexe 3 : Programmes de Codage et Décodage Run Length

Annexe 3.1 : Le Codage

```
program compress_long_run;

type parmtype = string[127];
     name_file = parmtype;
     filebyte = file of byte;

var filenam1,filenam2:name_file;
    pos_point:integer;
    car_special:byte;


procedure getparm(var s:parmtype);
          (* to obtain the command-line parameter
             MS-DOS DEPENDENT *)

var parms: parmtype absolute CSEG:$80;
begin
 s:='';
 while ((length(parms)>0) and (parms[1]=' ')) do delete(parms,1,1);
 while ((length(parms)>0) and (parms[1]<>' ')) do
       begin s:=s+parms[1];
             delete(parms,1,1)
       end
end;


procedure getnextcar(var file_in:filebyte;
                     var endf:boolean;
                     var car: byte);
(* procédure réalisant la lecture en avant
          - voir [JACKSON 1975] *)
(* si on se trouve à la fin de FILE_IN,
     ENDF a la valeur "TRUE" et CAR est indéterminé
   sinon
     ENDF n'est pas modifié, et CAR à la valeur du caractère
                                    courant de FILE_IN.   *)
begin
if not EOF(file_in) then read(file_in,car)
                    else endf:=TRUE
end;


procedure putnextcar(var file_out:filebyte;
                         endf:boolean;
                     var car: byte);
(*  si  ENDF n'a pas la valeur "TRUE",
     alors CAR est écrit  dans FILE_IN *)
begin
if not endf then write(file_out,car)
end;
```

```
procedure search_special_car(var filename:name_file;
                             var car_special: byte);

(* Cette procédure renvoie le caractère spécial (CAR_SPECIAL)
     sélectionné selon l'Algorithme 3.6 à partir du nom
     du fichier (FILENAME) contenant les données à compresser.
   L'algorithme peut sélectionner un quelconque caractère, à
     l'exception du caractère ASCII 026, marque de fin de fichier
     dans le système MS-DOS. *)

type filebyte = file of byte;


var file_in: filebyte; (* fichier à compresser *)
    tab_short_run: array[0..255] of real; (* correspond à la
                                             table T de
                                             l'algorithme 3.6 *)
    endf:boolean; (* ENDF a la valeur "TRUE" si CAR contient
                     une valeur significative après l'exécution de
                     GETNEXTCAR ; sinon  ( on se trouvait à la fin
                     du fichier ) ENDF a la valeur "FALSE" *)
    car,           (* caractère courant *)
     oldcar,       (* caractère du groupe courant *)
     length:byte;  (* longueur du groupe courant *)
    i,             (* position courante dans la table *)
     pos:integer;  (* position du minimum courant *)
    min:real;      (* valeur du minimum courant *)

begin
     for i:=0 to 255 do tab_short_run[i]:=0.0;
     assign(file_in,filename);
     reset(file_in);
     getnextcar(file_in,endf,car); (* lecture en avant *)
     while not endf do
     begin
          length:=0;
          oldcar:=car;
          repeat    (* lecture d'un groupe *)
                length:=length+1;
                getnextcar(file_in,endf,car);
          until ((car<>oldcar) or endf or (length=255));
          if (length<4)
          then (* constitution de la table *)
               tab_short_run[car]:=tab_short_run[car]+1.0
     end;
     close(file_in);
     pos:=0;
     min:=1.0e+20; (* "grand" chiffre *)
     for i:=0 to 255 (* recherche du minimum *)
     do if ((tab_short_run[i]<min) and (i<>26)) then pos:=i;
          (* ne pas utiliser le caractère ^Z (EOFile, ASCII 26)
             comme caractère spécial *)
     car_special:=pos
end;
```

```
procedure process_run(var file_in:filebyte;
                      var file_out:filebyte;
                      var endf:boolean;
                      var car:byte;
                          car_special: byte);

(* traite un groupe de caractères *)

var length:byte;
    oldcar:byte;
    i:integer;

begin
     length:=0;
     oldcar:=car;
     repeat (* lecture d'un groupe *)
           length:=length+1;
           getnextcar(file_in,endf,car);
     until ((car<>oldcar) or endf or (length=255));
     case length of
          1,2,3: for i:=1 to length
                    do putnextcar(file_out,endf,oldcar);
          26:begin  (* ce cas additionnel reflète la
                       nécessité d'éviter l'insertion du
                       caractère spécial EOFile (ASCII 026) ;
                       dans ce cas nous séparons la chaîne de
                       26 caractères en deux chaînes
                       plus petites *)
                    putnextcar(file_out,endf,car_special);
                    putnextcar(file_out,endf,oldcar);
                    length:=25;
                    putnextcar(file_out,endf,length);
                    putnextcar(file_out,endf,oldcar)
             end;
          else begin
               putnextcar(file_out,endf,car_special);
               putnextcar(file_out,endf,oldcar);
               putnextcar(file_out,endf,length)
               end
     end (* of case *)
end;
```

```
procedure compress_run
      (filenam1,filenam2:name_file;
       car_special:byte);

(* implémente l'algorithme 3.2 *)

var file_in, file_out:filebyte;
    i:integer;
    endf:boolean;
    car,car2,oldcar:byte;

begin
     assign(file_in,filenam1);
     assign(file_out,filenam2);
     reset(file_in);
     rewrite(file_out);
     endf:=FALSE;
     getnextcar(file_in,endf,car); (* lecture en avant *)
     if not endf then putnextcar(file_out,endf,car_special);
     (* la valeur de ce caractère doit se trouver en début
          de fichier pour le décodage *)
     while not endf do
       process_run(file_in,file_out,endf,car,car_special);
     close(file_in);
     close(file_out)
end;


begin (* of main program *)
      getparm(filenam1);
      pos_point:=pos('.',filenam1);
      if pos_point=0 then filenam2:=concat(filenam1,'.run')
                     else begin
                          filenam2:=copy(filenam1,1,pos_point-1);
                          filenam2:=concat(filenam2,'.run')
                          end;
     (* this is to obtain the names of the files --MS-DOS DEPENDENT *)
     search_special_car(filenam1,car_special);
     compress_run(filenam1,filenam2,car_special)
end.
```

Annexe_3.2_:_Le_Décodage

```
program decompress_long_run;

type parmtype = string[127];
     name_file = parmtype;
     filebyte = file of byte;

var filenam1,filenam2:name_file;
    pos_point:integer;



procedure getparm(var s:parmtype);
          (* to obtain the command-line parameter
             MS-DOS DEPENDENT *)

var parms: parmtype absolute CSEG:$80;
begin
 s:='';
 while ((length(parms)>0) and (parms[1]=' ')) do delete(parms,1,1);
 while ((length(parms)>0) and (parms[1]<>' ')) do
       begin s:=s+parms[1];
             delete(parms,1,1)
       end
end;



procedure getnextcar(var file_in:filebyte;
                     var endf:boolean;
                     var car: byte);

begin
if not EOF(file_in) then read(file_in,car)
                    else endf:=TRUE
end;



procedure putnextcar(var file_out:filebyte;
                         endf:boolean;
                     var car: byte);

begin
if not endf then write(file_out,car)
end;
```

```
procedure read_a_group(var file_in:filebyte;
                       var endf:boolean;
                       var car,length:byte;
                           car_special:byte);
begin
     getnextcar(file_in,endf,car);
     if not endf
     then if (car=car_special)
          then begin
                    getnextcar(file_in,endf,car);
                    getnextcar(file_in,endf,length)
               end
          else length:=1
end;


procedure decompress_run(filenam1,filenam2:name_file);

(* implémente l'algorithme 3.4 *)

var file_in, file_out : filebyte;
    i : integer;
    endf : boolean;
    car,car_special,length : byte;

begin
     assign(file_in,filenam1);
     assign(file_out,filenam2);
     reset(file_in);
     rewrite(file_out);
     endf:=FALSE;
     read_a_group(file_in,endf,car,length,car_special);
          (* lecture en avant d'un groupe *)
     while not endf
     do begin
             for i:=1 to length do putnextcar(file_out,endf,car);
             read_a_group(file_in,endf,car,length,car_special)
        end;
     close(file_in);
     close(file_out)
end;


begin (* of main program *)
      getparm(filenam1);
      pos_point:=pos('.',filenam1);
      if pos_point=0 then filenam2:=concat(filenam1,'.dcr')
                     else begin
                          filenam2:=copy(filenam1,1,pos_point-1);
                          filenam2:=concat(filenam2,'.dcr')
                          end;
     (* this is to obtain the names of the files --MS-DOS DEPENDENT *)
     decompress_run(filenam1,filenam2)
end.
```

Annexe 4 : Programmes de Codage et Décodage Huffman

Annexe 4.1 : Le Codage

```
program compression_huffman;

type parmtype = string[127]; (* command-line parameters *)

     name_file = parmtype;    (* the name of the file
                                 to be compressed *)

     t_el_tab_freq            (* one element of the "tree" *)
                 = record
                   car : integer;
                   freq_car : real;
                   left,right:integer (* pointers to the left and
                                         the right elements of
                                         the "tree" *)
                  end;


     t_tab_freq = array[0..255] of t_el_tab_freq;

     t_tab_to_sort = t_tab_freq;

     t_code_huff = array[0..31] of byte;
                      (* worst case, 32*8 = 256 bits for
                         one character *)

     t_el_tab_code = record
                       car: byte;
                         (* character concerned *)
                       prefixe_length: byte;
                       length: byte;
                         (*
                            if prefixe_length = 0
                            then  code = (length+1) bits
                              ( because 0 ≤ length ≤ 255)
                                    => (1 ≤ bits ≤ 256)
                            else   (<=> prefixe_length = 1)
                                  length is undefined
                                  and code is empty
                                                    *)
                       code_huff: t_code_huff
                     end;

     t_tab_code = array[0..255] of t_el_tab_code;
```

```
var filenam1, (* name of the input file *)

    filenam2: (* name of the output file *)

              name_file;

    total: real; (* number of characters in the input file *)

    tab_freq: t_tab_freq;
          (* table of the frequency of each character of the
             input file; the "frequency" may be the frequency
             in a mathematical sense, or the number of
             occurences of the character in the input file *)

    tab_code: t_tab_code;
          (* table of huffman codes of each character *)

    position_point:integer;
          (* position of the point in the name of the
             input file -- MS-DOS DEPENDENT --          *)


procedure getparm(var s:parmtype);

(* procedure to get the name of the input file from
   the command line -- MS-DOS DEPENDENT -- *)

var parms: parmtype absolute CSEG:$80;

begin
     s:='';
     while ((length(parms)>0) and (parms[1]=' '))
     do delete(parms,1,1);
     while ((length(parms)>0) and (parms[1]<>' '))
     do begin
              s:=s+parms[1];
              delete(parms,1,1)
        end
end;
```

```
procedure calcule_frequences(filename:name_file;
                             var total:real;
                             var tab_freq: t_tab_to_sort);

(* this procedure
     receives the name of a file : FILENAME
     outputs the number of caracters of the file : TOTAL
             a table of the frequencies and corresponding
             caracters, in ascending order of frequencies,
             of the caracters of the file : TAB_FREQ *)

var file_in:file of byte;

    car:byte;

    i:byte;


(*-------------------------------------------------------------*)
procedure quicksort(var a:t_tab_to_sort;binf,bsup:integer);
(*-------------------------------------------------------------*)

(* this is a straightforward implementation of HOARE's quicksort
algorithm, as described in his paper (C.A.R.Hoare, "Quicksort"
Computer Journal,5,1(1962)) ; another good (an youngest)
description, with variations, is provided in chapter 9 of
[SEDGEWICK 1983] ; it must be said that attention has be
provided to use a non-recursive version of this algorithm,
which could be in this case adapted to all language in a
(not far from) direct way ...*)

var i,j,l,r: integer;

    v,t: t_el_tab_freq;

    stack: array[0..50] of integer;

    p: integer;


begin
     l:=binf; r:=bsup; p:=2;
     repeat
           if (r>l)
           then begin
(* This       *)    v:=a[r]; i:=l-1; j:=r;
(* correspond *)    repeat
(* to the     *)          repeat i:=i+1
(*            *)          until (a[i].freq_car>=v.freq_car);
(* procedure  *)          repeat j:=j-1
(*            *)          until (a[j].freq_car<=v.freq_car);
(* PARTITION  *)          t:=a[i];a[i]:=a[j];a[j]:=t
(*            *)    until (j<=i);
(*            *)    a[j]:=a[i];a[i]:=a[r];a[r]:=t;
```

```
                    if ((i-l) > (r-i))
                    then begin
(* PUSH the     *)              stack[p]:=l;
(* left part    *)              stack[p+1]:=i-1;
(* on the stack *)              l:=i+1
                         end
                    else begin
(* PUSH the     *)              stack[p]:=i+1;
(* right part   *)              stack[p+1]:=r;
(* on the stack *)              r:=r-1
                         end
               end
          else begin
(*              *)
(* POP the stack *)  p:=p-2; l:=stack[p]; r:=stack[p+1]
(*              *)
               end
     until (p=0)
end;



begin
     assign(file_in,filename);
     reset(file_in);
     for i:=0 to 255 do (* initialization of the table *)
     begin
          tab_freq[i].car:=i;
                    (* car of element i = i *)
          tab_freq[i].freq_car:=0.0;
                    (* initial number of car i in the file *)
          tab_freq[i].left:=-1;
          tab_freq[i].right:=-1
                    (* pointer = -1 <==> points to nothing *)
     end;
     total:=0;
     while not EOF(file_in) do
     begin
          read(file_in,car);
          total:=total+1;
          tab_freq[car].freq_car:=tab_freq[car].freq_car + 1
     end;
     close(file_in);
     quicksort(tab_freq,0,255);
     if total>0
     then for i:=0 to 255 do
           tab_freq[i].freq_car:=tab_freq[i].freq_car/total
     (* freq_car contains the mathematical frequency
        of the car in the file *)
end;
```

```
(*=================================================================*)
(* Main procedure to create a table of huffman codes              *)
(* from a table of caracters and their frequencies                *)
(*=================================================================*)

procedure etablit_code(tab_freq: t_tab_freq;
                       var tab_code: t_tab_code);

type t_tree = array[0..510] of t_el_tab_freq;
     (* tree = 256 original elements + 255 intermediates
             = 256 leaves + 255 nodes                  *)

     t_ptr_below_tree = array[0..255] of integer;
     (* see expression 4.4 *)


var b_inf, b_sup :integer;

    tree:t_tree;

    ptr_below_tree: t_ptr_below_tree;

    i, root: integer;

    code_current: t_code_huff;


(*-----------------------------------------------------------------
subroutines to manipulate bits in a Huffman code
             (array[0..31] of byte) = 256 bits
-----------------------------------------------------------------*)

function testset(to_test:t_code_huff;pos:integer):boolean;

(* return TRUE if the POSth bit of to_test is set to 1
        FALSE in the other case *)

var temp:integer;

    posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);


begin
     posbit:=7-(pos mod 8);
     (* because in a byte,
              bit 0 as a weight of 128
                  1                 64
                     ...                *)
     pos_char:=(pos div 8);
     temp:=to_test[pos_char];
     temp:=temp AND Integer(expos[posbit]);
     testset:=(temp > 0)
end;
```

```
procedure setbit(var code_huff:t_code_huff;
                 var oldval:byte;
                     pos:integer);
(* if the POSth bit of CODE_HUFF = 0 then set it to 1
                                     else  do nothing
     return the old value of the bit in OLDVAL *)

var posbit, pos_char:byte;
const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     if not testset(code_huff,pos)
        then begin
             posbit:=7-(pos mod 8);
                  (* because in a byte,
                bit 0 as a weight of 128
                    1                64
                        ...             *)
             pos_char:=(pos div 8);
             code_huff[pos_char]:=code_huff[pos_char]
                                  + expos[posbit];
             oldval:=0
             end
        else oldval:=1
end;


procedure unsetbit(var code_huff:t_code_huff;
                   var oldval:byte;
                       pos:integer);
(* if the POSth bit of CODE_HUFF = 1 then set it to 0
                                     else  do nothing
     return the old value of the bit in OLDVAL *)

var posbit, pos_char:byte;
const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     if testset(code_huff,pos)
        then begin
             posbit:=7-(pos mod 8);
                  (* because in a byte,
                bit 0 as a weight of 128
                    1                64
                        ...             *)
             pos_char:=(pos div 8);
             code_huff[pos_char]:=code_huff[pos_char]
                                      - expos[posbit];
             oldval:=1
             end
        else oldval:=0
end;
(*---------------------------------------------------------
end of subroutines to manipulate bits in a huffman code
-------------------------------------------------------------*)
```

```
(*-------------------------------------------------
subroutine to initialize the tree
-------------------------------------------------*)

procedure init_tree(var tree: t_tree;
                        tab_freq:t_tab_freq;
                    var b_inf,b_sup:integer;
                         (* at the end of this procedure,
                            b_inf = number of the pointer
                                    pointing to the 1st
                                    character of freq. > 0
                                  = 256 if it doesn't exist
                            b_sup = always 255 *)
                    var ptr_below_tree: t_ptr_below_tree);
                            (* at the end,
                               the pointers point to the 256
                               elements of the table,
                               the leaves *)

var i :integer;

begin
     (* initialization of the tree with the 256 leaves ...
        and computation of the position of the 1st element with
        a non-zero frequency *)
     i:=0;
     while ((i<=255) and not (tab_freq[i].freq_car>0.0))
       do  i:=i+1;
         (* to find the 1st element with a non-zero frequency,
            = 256 if there isn't                                  *)
     b_inf:=i;
     b_sup:=255;
     for i:=0 to 255 do
     begin
          ptr_below_tree[i]:=i;
          tree[i]:=tab_freq[i]
     end;
     (* initialization of the 255 nodes *)
     for i:=256 to 510
     do begin
          tree[i].car := 0;
             (* the car has no meaning for the
                element of the tree which are not leaves *)
          tree[i].freq_car := 0.0;
          tree[i].left := -1;
          tree[i].right := -1
             (* left and right pointers point to nothing
                at the beginning ---> value = -1          *)
        end
end;
```

```
(*-------------------------------------
subroutine to create the tree
-------------------------------------*)

procedure create_tree(var tree:t_tree;
                      var b_inf,b_sup: integer;
                      var ptr_below_tree: t_ptr_below_tree;
                      var root:integer);

var next:integer; (* points to the next element free for building
                      the tree (equivalent to the PASCAL standard
                      function NEW, for "pointer" in pascal
                      sense) *)


(*=====================================
subroutine to create a node
=====================================  *)

procedure create_node(ptr_to_use: integer;
                  var b_inf,b_sup: integer;
                  var tree: t_tree;
                  var ptr_below_tree: t_ptr_below_tree);

var i, next, place_to_insert, temp:integer;

begin
(*        the element to create has a frequency which is the
     sum of the frequencies of the two elements
     tree[ptr_below_tree[b_inf]] & tree[ptr_below_tree[b_inf+1]];
          the left "pointer" of this newly created element
     contains the "address" of tree[ptr_below_tree[b_inf]],
     which is a position in the table, and is equal to
     ptr_below_tree[b_inf];
          the right "pointer" of this newly created element
     contains the "address" of tree[ptr_below_tree[b_inf+1]],
     which is a position in the table, and is equal to
     ptr_below_tree[b_inf+1];
          BINF is incremented by one, and the "pointer" at the
     BINFth position now points to the newly created element,
     but the sorted order no longer exists ! *)

    tree[ptr_to_use].freq_car :=
        tree[ptr_below_tree[b_inf] ].freq_car +
        tree[ptr_below_tree[b_inf + 1] ].freq_car;
    tree[ptr_to_use].left:=ptr_below_tree[b_inf];
    tree[ptr_to_use].right:=ptr_below_tree[b_inf+1];
    b_inf:=b_inf+1;
    ptr_below_tree[b_inf]:=ptr_to_use;
    i:=b_inf+1;
```

```
    (* this "while" loop search for the place to insert the
     new element to conserve the ascending order of frequencies
     in the table *)
    while ((i <= b_sup)
     and (tree[ptr_below_tree[b_inf]].freq_car
           > tree[ptr_below_tree[i]].freq_car))
    do i:=i+1;
    place_to_insert:=i-1;

    (* the insertion is made now *)
    temp:=ptr_below_tree[b_inf];
    for i:=b_inf to (place_to_insert-1)
      do ptr_below_tree[i]:=ptr_below_tree[i+1];
    ptr_below_tree[place_to_insert]:=temp
end;

(*================================================
end of the subroutine to create a node
================================================*)


begin (* of the subroutine to create the tree *)
     next:=256; (* points to the first "free" place
                   in the table *)
     while (b_inf<b_sup) do
     begin
          create_node(next,b_inf,b_sup,tree,ptr_below_tree);
          next:=next+1
     end;
     if (b_inf=b_sup)
     then root:=ptr_below_tree[b_inf]
          (* because the root is the last element created *)
     else root:=-1
          (* b_inf > b_sup
          (<--> during the initialization of the tree,
          no characters with a frequency > 0) --> tree is empty
          (<--> root = -1)     *)
end;

(* ------------------------------------
end of the subroutine to create the tree
     ------------------------------------*)
```

```
(* ------------------------------------------------
subroutine to initialize the code of each character
------------------------------------------------*)

procedure init_code(var tab_code:t_tab_code);

(* for each element of the table of codes :
   - give to car its value
   - set the prefix to 1 <-> there is no code for this element
   - set length of the code to 0
     (this is not very important here, because prefix=1
          => there is no code
   - set the 256 bits of the code to 0
     (this is also not important here)
*)

var i,j: integer;


begin
     for i:=0 to 255 do
     begin tab_code[i].car:=i;
           tab_code[i].prefixe_length := 1;
           tab_code[i].length:=0;
           for j:=0 to 31 do tab_code[i].code_huff[j]:=0
     end
end;


(* ------------------------------------------
recursive subroutine to create the codes
   ---------------------------------------*)

procedure encode(var tree:t_tree;
                 el_current:t_el_tab_freq;
                 depth:integer;
                 var code_current:t_code_huff;
                 var tab_code:t_tab_code);


var oldval:byte; (* for the memorisation of the old value
                    of the bit, which must be restored *)
```

```
begin
     if ((el_current.left=-1) and (el_current.right=-1))

     then begin
               (* the current element is a leave *)
               tab_code[el_current.car].car:=el_current.car;
               tab_code[el_current.car].prefixe_length:=0;
               tab_code[el_current.car].length:=depth;
               tab_code[el_current.car].code_huff:=code_current
          end

     else begin
               (* the current element is a node
                    we must

                  ¦ - unset the bit
     left part of + - encode the left subtree
      the tree    ¦ - restore the old value of the bit

                  ¦ - set the bit
    right part of + - encode the right subtree
      the tree    ¦ - restore the old value of the bit
                                                    *)

               unsetbit(code_current,oldval,depth+1);
               encode(tree,tree[el_current.left],depth+1,
                      code_current,tab_code);
               if (oldval=1)
                  then setbit(code_current,oldval,depth+1);

               setbit(code_current,oldval,depth+1);
               encode(tree,tree[el_current.right],depth+1,
                      code_current,tab_code);
               if (oldval=0)
                  then unsetbit(code_current,oldval,depth+1);
          end

(* the suppression of the recursivity here is difficult, even if
     possible, because there are TWO recursive calls ;
     of course, this must be done for the implementation
     in Assembly language...*)

end;
```

```
(*============================================================*)
(*                                                            *)
(*        main routine for creating the code                  *)
(*                                                            *)
(*============================================================*)


begin
     init_tree(tree,tab_freq,b_inf,b_sup,ptr_below_tree);
     create_tree(tree,b_inf,b_sup,ptr_below_tree,root);
     init_code(tab_code);
     if (root>-1)     (* if root = -1 : the tree is empty *)
     then begin
          for i:=0 to 31 do code_current[i]:=0;
          if ((tree[root].left=-1) and (tree[root].right=-1))
          then tab_code[tree[root].car].prefixe_length:=0
               (* this is a very special case : there is only
                  one element in the table (<=> the file is
                  composed of the same character); the only
                  thing to do is to give to this element the
                  code 0, with prefix=0 (there is a code for
                  this element), and with a length of 0 (which
                  means that the length of the code is 1)    *)
          else encode(tree,tree[root],-1,code_current,tab_code)
          end;
end;
```

```
procedure file_to_file_comp(filenam1, (* name of input file *)
                            filenam2  (* name of output file *)
                                      :name_file;
                            total     (* number of char. in
                                         the input file *)
                                      : real;
                            tab_code (* table of Huffman codes *)
                                      :t_tab_code);

type filebyte = file of byte;

var file_in, file_out: filebyte;

    size_overhead:integer;

    size_int: integer;

    size_byte,i,j,pointer,car_to_code,bufcode: byte;

    var tot_car : record
                  case tot_car_or_6_byte: boolean of
                     TRUE: (tot_car_sel:real);
                     FALSE: (tot_6_byte_sel: array[1..6] of byte)
                  end;
     (* this is a VARIANT RECORD, which is a pascal "trick"
        to transform a real (here, 6 bytes) in the 6
        corresponding bytes;
        -- MS-DOS TURBO-PASCAL IMPLEMENTATION DEPENEDENT -- *)

(*============================================================*)

procedure encode (* procedure to encode a byte *)
          (car_to_code (* the character to code *)
                       : byte;
       var bufcode (* the buffer which will be written - because
                      you can't write a file bit per bit ...*)
                       : byte;
       var file_out    :filebyte;
           tab_code (* the table of codes *)
                       :t_tab_code;
       var pointer (* indication of the position in the "buffer"
                      if pointer = 8 then write buffer
                                          set pointer to 0 *)
                       :byte);

var i: byte;
```

```
(* ==========================================================
subroutines (local to the encode procedure) to manipulate bits
  in a huffman code
  in a 8 bits buffer
========================================================== *)

function testset(to_test:t_code_buff;pos:integer):boolean;

(* return TRUE if the POSth bit of to_test is set to 1
          FALSE in the other case *)

var temp:integer;
    posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);


begin
     posbit:=7-(pos mod 8);
     (* because in a byte,
                bit 0 as a weight of 128
                    1                 64
                              ...                 *)
     pos_char:=(pos div 8);
     temp:=to_test[pos_char];
     temp:=temp AND Integer(expos[posbit]);
     testset:=(temp > 0)
end;



procedure setbit(var buffer:byte; pos:byte);
(* set the bit at the POSth position of buffer to 1 *)
(* positions in a byte : 01234567                    *)

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);


begin
     buffer:=buffer + expos[7-pos];
end;

(* ==========================================================
end of subroutines (local to the encode procedure) to manipulate
 bits   in a huffman code
        in a 8 bits buffer
========================================================== *)
```

```
begin
     for i:=0 to tab_code[car_to_code].length
     do begin
                if testset(tab_code[car_to_code].code_huff,i)
                then setbit(bufcode,pointer);
                pointer:=pointer+1;
                if pointer=8
                then begin (* the buffer must be flushed *)
                           pointer:=0;
                           write(file_out,bufcode);
                           bufcode:=0
                     end
        end
end;

(*-------------------------------------------------------------*)

begin
     assign(file_in,filenam1);    (* MS-DOS DEPENDENT *)
     assign(file_out,filenam2);   (* MS-DOS DEPENDENT *)
     size_overhead:=0;
     reset(file_in);
     rewrite(file_out);
     tot_car.tot_car_or_6_byte:=TRUE;
     tot_car.tot_car_sel:=total;
     tot_car.tot_car_or_6_byte:=FALSE;
     for i:=1 to 6 do write(file_out,tot_car.tot_6_byte_sel[i]);
     size_overhead:=size_overhead+6;
     (* the 6 first bytes of the packed file contain the number
        of characters in the original file; the 6 bytes must be
        transformed into a real with a
        RECORD VARIANT (CASE OF...) -- MS-DOS DEPENDENT        *)

     size_int:= 1; (* size_int from 0 to 255
                      for a table of 1 to 256 element *)
     for i:=0 to 255 do if (tab_code[i].prefixe_length=0)
                           then size_int:=size_int+1;
                     (* count the number of significant
                        entries in the table of huffman codes *)
     size_byte:=size_int;
     write(file_out,size_byte);
     size_overhead:=size_overhead+1;
     for i:=0 to 255 do
         if (tab_code[i].prefixe_length=0)
         then begin
                   write(file_out,tab_code[i].car);
                   write(file_out,tab_code[i].length);
                   size_overhead:=size_overhead+2;
                   for j:=0 to (tab_code[i].length DIV 8)
                   do begin
                        write(file_out,tab_code[i].code_huff[j]);
                        size_overhead:=size_overhead+1
                      end
              end;
```

```
(*
this is the structure of the packed file :
------------------------------------------
byte 1 to 6 : size of the original file;

byte 7 : number of entries in the huffman code table
      0 ≤ byte 7 ≤ 255 -> 1 ≤ number of entries ≤ 256;

see figure 4.5 for the structure of the code table *)


     pointer: 0;
     bufcode:=0;
     while not EOF(file_in)
     do begin
             read(file_in,car_to_code);
             encode(car_to_code,bufcode,file_out,tab_code,pointer)
        end;
     if pointer > 0 then write(file_out,bufcode);
     close(file_in);
     close(file_out)
end;



begin
     getparm(filenam1);
     position_point:=pos('.',filenam1);
     if position_point=0
     then filenam2:=concat(filenam1,'.huf')
     else begin
               filenam2:=copy(filenam1,1,position_point-1);
               filenam2:=concat(filenam2,'.huf')
          end;
     (* this is to obtain the external file of the file
          to compress, and the compressed file;
          this is MS-DOS DEPENDENT *)

     calcule_frequences(filenam1,total,tab_freq);
     (* computation of the frequency of each element
                    and the size of the input file   *)

     etablit_code(tab_freq,tab_code);
     (* construction of the code *)

     file_to_file_comp(filenam1,filenam2,total,tab_code)
     (* coding of the file *)
end.
```

Annexe_4.2_:_Le_Décodage

```
program decompact_huffman;

type filebyte = file of byte;

     parmtype = string[127];

     name_file = parmtype;

     tree_ptr = integer; (* may be in the range of
                            -1..510 *)

     t_el_tree = record
                  car : byte;
                  left, right:tree_ptr;
                  (* we don't need the frequency here *)
                 end;

     t_tree = array[0..510] of t_el_tree;

     t_code_huff = array[0..31] of byte;
                      (* worst case : code = 32*8 = 256 bits
                         for one character *)

     t_el_tab_code = record
                       car: byte;
                         (* character concerned *)
                       prefix_length: byte;
                       length: byte;
                         (* if prefix = 0
                            then if length = x : code = (x+1) bits
                              (0 s length s 255)
                                    => (1 s bits s 256)
                            if prefix = 1 :
                            then length is undefined
                                 and code is empty          *)
                       code_huff: t_code_huff
                      end;

     t_tab_code = array[0..255] of t_el_tab_code;


var filenam1,filenam2:name_file;

    position_point:integer;
```

```
(*-------------------------------------------------------
subroutines to manipulate bits in
a Huffman code (array[0..31] of byte) = 256 bits
-------------------------------------------------------*)

function testset(to_test:t_code_huff;pos:integer):boolean;
(* return TRUE if the POSth bit of to_test is set to 1
          FALSE in the other case *)

var temp:integer;

    posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     posbit:=7-(pos mod 8);
     (* because in a byte,
                bit 0 as a weight of 128
                    1                  64
                              ...          *)
     pos_char:=(pos div 8);
     temp:=to_test[pos_char];
     temp:=temp AND Integer(expos[posbit]);
     testset:=(temp > 0)
end;


procedure setbit(var code_huff:t_code_huff;
                 var oldval:byte;
                     pos:integer);
(* if the POSth bit of CODE_HUFF = 0 then set it to 1
                                    else  do nothing
return the old value of the bit in OLDVAL *)

var posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     if not testset(code_huff,pos)
        then begin
             posbit:=7-(pos mod 8);
                  (* because in a byte,
                bit 0 as a weight of 128
                    1                  64
                              ...          *)
             pos_char:=(pos div 8);
             code_huff[pos_char]:=code_huff[pos_char]
                                  + expos[posbit];
             oldval:=0
             end
        else oldval:=1
end;
```

```
procedure unsetbit(var code_huff:t_code_huff;
                   var oldval:byte;
                       pos:integer);
(* if the POSth bit of CODE_HUFF = 1 then set it to 0
                                    else  do nothing
return the old value of the bit in OLDVAL *)

var posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     if testset(code_huff,pos)
        then begin
             posbit:=7-(pos mod 8);
                  (* because in a byte,
                     bit 0 as a weight of 128
                       1                  64
                            ...               *)
             pos_char:=(pos div 8);
             code_huff[pos_char]:=code_huff[pos_char]
                                  - expos[posbit];
             oldval:=1
             end
        else oldval:=0
end;


(* ------------------------------------------------------------
end of subroutines to manipulate bits in a huffman code
------------------------------------------------------------*)


procedure getparm(var s:parmtype);

(* -- MS DOS DEPENDENT -- to get the command-line parameters *)

var parms: parmtype absolute CSEG:$80;

begin
     s:='';
     while ((length(parms)>0) and (parms[1]=' '))
     do delete(parms,1,1);
     while ((length(parms)>0) and (parms[1]<>' '))
     do begin s:=s+parms[1];
              delete(parms,1,1)
        end
end;
```

```
procedure buildleaf(code:t_el_tab_code;
                    var tree:t_tree;
                    var next: tree_ptr);

(* build a leaf of the tree *)

var i:byte;

    cour: tree_ptr;


begin
     i:=0;
     cour:=0;
     while (i<=code.length)
     (* create a path until the leaf *)
     do begin
              if testset(code.code_huff,i)
              then begin
                        (* go to the right subtree *)
                        if (tree[cour].right=-1)
                        then begin
                                  (* there is no subtree *)
                                  tree[cour].right:=next;
                                  (* create the right subtree *)
                                  next:=next+1
                             end;
                        cour:=tree[cour].right
                      end
              else begin
                        (* go to the left subtree *)
                        if (tree[cour].left=-1)
                        then begin
                                  (* there is no subtree *)
                                  tree[cour].left:=next;
                                  (* create the left subtree *)
                                  next:=next+1
                             end;
                        cour:=tree[cour].left
                   end;
              i:=i+1
        end;
     (* build the leaf *)
     tree[cour].car:=code.car
end;
```

```
procedure buildtree(     tab_code:t_tab_code;
                     var tree:t_tree);

(* build the complete tree *)


var i:byte;

    next:integer;

    j:integer;


begin
     for j:=0 to 510 do
     (* initialization of the "heap" of intermediate leaves *)
     begin
          tree[j].car:=0;
          tree[j].right:=-1;
          tree[j].left:=-1
     end;
     next:=1;
     (* to simulate a dynamic allocation of the
     intermediate leaves *)
     for i:=0 to 255
     (* with each code : build the corresponding path in
        the tree,
        with bit 0 means "go to the left"
             bit 1 means "go to the right" *)
     do if (tab_code[i].prefix_length=0)
        then buildleaf(tab_code[i], tree,next)
end;


procedure decode(var file_in:filebyte;
                 var car_to_decode: byte;
                 var bufcode: byte;
                     tree: t_tree;
                 var pointeur: byte);

(* procedure to decode one car of the original file *)

var cour: tree_ptr;
```

```
function testsetbyte(to_test,pos:byte):boolean;

(* return TRUE if the POSth bit of to_test is set to 1
          FALSE in the other case *)

var temp:integer;

    posbit, pos_char:byte;

const expos:array[0..7] of byte = (1,2,4,8,16,32,64,128);

begin
     posbit:=7-pos;
     (* because in a byte,
                bit 0 as a weight of 128
                    1                  64
                           ...              *)
     temp:=to_test;
     temp:=temp AND Integer(expos[posbit]);
     testsetbyte:=(temp > 0)
end;


begin
     cour:=0;
     while not((tree[cour].left=-1) or (tree[cour].right=-1))
     (* when they will be = to -1 : we will be at the bottom of
        the tree *)
     do begin
        if (pointeur>7) (* the all buffer has been read : we must
                           read the following 8 bits *)
           then begin
                     read(file_in,bufcode);
                     pointeur:=0
                end;
        if testsetbyte(bufcode,pointeur)
           then cour:=tree[cour].right
           else cour:=tree[cour].left;
        pointeur:=pointeur+1
     end;
     (* we are at a leaf of the tree : we have the character *)
     car_to_decode:=tree[cour].car
end;
```

```
procedure file_comp_to_file(filenam1, filenam2:name_file);

type var_record_tag = (cons_6_byte, cons_real);

var file_in,
    file_out: filebyte;

    total,k:real;

    size_tab: byte;

    i,j,
    pointeur,
    car_to_decode,
    bufcode,
    car_concerned: byte;


    (* the following VARIANT RECORD is a trick to transform
       a real number into the corresponding 6 bytes; this is
       MS-DOS DEPENDENT                                        *)
    tot_car : record
                  case tot_car_or_6_byte: var_record_tag of
                     cons_real: (tot_car_sel:real);
                     cons_6_byte: (tot_6_byte_sel:
                                    array[1..6] of byte)
                  end;

    tree: t_tree;

    tab_code:t_tab_code;


begin
     assign(file_in,filenam1);
     assign(file_out,filenam2);
     reset(file_in);
     tot_car.tot_car_or_6_byte:=cons_6_byte;
     for i:=1 to 6 do read(file_in,tot_car.tot_6_byte_sel[i]);
     (* the 6 first bytes of the input file contains the number
        of characters in the original file, which must be transform
        into a real                                             *)
     tot_car.tot_car_or_6_byte:=cons_real;
     total:=tot_car.tot_car_sel;
```

```
      read(file_in,size_tab);
           (* size_tab from 0 to 255
              for a table from 1 to 256 element *)
      for i:=0 to 255 do
      (* initialization of the codes *)
       begin
          tab_code[i].car:=i;
          tab_code[i].prefix_length:=-1;
          tab_code[j].length:=0;
          for j:=0 to 31 do tab_code[i].code_huff[j]:=0;
       end;
      for i:=0 to size_tab do (* read a code *)
      begin
           read(file_in,car_concerned);
           tab_code[car_concerned].car:=car_concerned;
           tab_code[car_concerned].prefix_length:=0;
           read(file_in,tab_code[car_concerned].length);
           for j:=0 to (tab_code[car_concerned].length DIV 8)
             do read(file_in,tab_code[car_concerned].code_huff[j])
      end;
      buildtree(tab_code,tree);
      rewrite(file_out);
      pointeur:=8;
      bufcode:=0;
      k:=1.0;
      while (k<=total) do
      begin
           decode(file_in,car_to_decode,bufcode,tree,pointeur);
           write(file_out,car_to_decode);
       k:  =k+1.0
      end;
      close(file_in);
      close(file_out)
end;


begin (* main program *)
      getparm(filenam1);
      position_point:=pos('.',filenam1);
      if position_point=0
      then filenam2:=concat(filenam1,'.dch')
      else begin
                filenam2:=copy(filenam1,1,position_point-1);
                filenam2:=concat(filenam2,'.dch')
           end;
      (* this is to obtain the names of the files
           --MS-DOS DEPENDENT *)
      file_comp_to_file(filenam1,filenam2)
end.
```

Annexe 5 : Programmes de Codage et Décodage LOVER

Annexe 5.1 : Le Codage

```
program code_delta;

(* la description du principe de base de l'algorithme et la
     spécification des procédures et fonctions est donnée
     en chapitre 6 de ce mémoire *)

const special_car : byte = 145; (* caractère 'æ' inhabituel *)
const N = 2048;          (* taille maximum d'un record *)

type parmtype = string[127];
     typstring = record          (* type de base *)
                 longueur: integer;
                 chaine : array[1..N] of byte
                   end;
     texte  =  file  of  byte;  (* type  des  fichiers
                                 - la structure des records
                                   est simulée au sein des
                                   procédures Get_typstring
                          et Put_typstring *)

var file_in: texte;              (* fichier à coder *)
    file_out: texte;             (* fichier résultat *)
    pos_point:integer;           (* pour la construction du nom
                                    du fichier résultat *)
    filenam1,filenam2:parmtype; (* nom des deux fichiers *)
    str1, str2:typstring;        (* les deux strings
                                    de travail *)
    OK : boolean;                (* si OK=FAUX : le record est
                                                 trop court *)


procedure getparm(var s:parmtype);
(* lit le nom du fichier à coder, donné avec l'appel
     du programme dans la ligne de commande *)
var parms: parmtype absolute CSEG:$80; (* MS-DOS dependent !!!!
                                          addresse absolue de la
                                          seconde partie de la
                                          ligne de commande *)
begin
     s:='';
     while ((length(parms)>0) and (parms[1]=' '))
     do delete(parms,1,1);
     while ((length(parms)>0) and (parms[1]<>' '))
     do begin s:=s+parms[1];
              delete(parms,1,1)
        end
end;
```

```
function length_string(str: typstring):integer;
(* correspond la fonction Length_string (voir 6.2) *)
begin
     length_string:=str.longueur
end;


procedure set_length_String(var Str:typstring;
                            Length:integer;
                            var OK:Boolean);
(* correspond à la procédure Set_length_string (voir 6.2) *)
begin
     if (Length<=N)
     then begin Str.longueur:=Length;
                OK:=TRUE
          end
     else OK:=FALSE
end;


procedure get_typstring(var file_in:texte;
                        var str:typstring;
                        var OK:boolean);
(* procédure Get_string (6.2) *)
(* la simulation d'un record se fait par la recherche d'un
     caractère LineFeed (ASCII 10)   *)
var car:byte;

begin
     str.longueur:=0;
     car:=0;
     OK:=TRUE;
     while ((not EOF(file_in)) and (car<>10)
           and (str.longueur<N) and OK)
     do begin
             read(file_in,car);
             if (car=special_car)
             then OK:=FALSE;
             str.longueur:=str.longueur+1;
             str.chaine[str.longueur]:=car
        end;
     (* fin d'un record normalement marquée par CR-LF *)
     if (not EOF(file_in) and (car<>10))
     then OK:=FALSE;
          (* tableau trop petit *)
     if OK then str.longueur:=str.longueur-2
          (* ne pas mettre dans le record
             les caractères CR et LF *)
end;
```

```
procedure put_typstring(var file_out:texte; var str:typstring);
(* procédure Put_string (voir 6.2) *)
var i:integer;
const cr : byte = 13;
      lf : byte = 10;
begin
     for i:=1 to length_string(str) do
          write(file_out,str.chaine[i]);
     write(file_out,cr,lf)  (* fin du record *)
end;


function getcar(str:typstring;pos:integer):byte;
(* fonction Getcar (voir 6.2) *)
begin
     if (pos<=length_string(str))
     then getcar:=str.chaine[pos]
end;


procedure setcar(var str:typstring; pos:integer; car:byte);
(* procedure setcar (voir 6.2) *)
begin
     if (pos<=length_string(str))
     then str.chaine[pos]:=car
end;


procedure code(var str1, str2: typstring; special_car: byte);
(* voir 6.2 et algorithme 6.1 *)
var i:integer;
temp:typstring;
begin
     temp:=str2;
     i:=1;
     while ((i<=length_string(str1))
          and (i<=length_string(str2)))
     do begin
        if (getcar(str1,i)=getcar(Str2,i))
        then setcar(str2,i,special_car);
        i:=i+1
        end;
        str1:=temp
end;
```

```
(* procédure principale réalisant le codage LOVER voir (6.2) *)
begin
     getparm(filenam1);
     (* pour obtenir le nom du premier fichier *)
     pos_point:=pos('.',filenam1);
     if pos_point=0
     then filenam2:=concat(filenam1,'.LOV')
     else begin
          filenam2:=copy(filenam1,1,pos_point-1);
          filenam2:=concat(filenam2,'.del')
          end;
     (* construction du nom du deuxième fichier =
        nom du premier fichier + extension "LOV" *)
     assign(file_out,filenam2);
     assign(file_in,filenam1);
     reset(file_in);
     rewrite(file_out);
     set_length_string(str1,0,OK);
     (* mise à zéro de Str1 *)
     while (not EOF(file_in) and OK) do
     begin
          get_typstring(file_in,str2,OK);
          if OK
          then begin
                    code(str1,str2,special_car);
                    put_typstring(file_out,str2)
               end
          else writeln('perte d''information !');
     end;
     close(file_in);
     close(file_out)
end.
```

Annexe_5.2_:_Le_Décodage

```
program decompress_delta;

const special_car : byte = 145;
const N = 2048;

type parmtype = string[127];
     typstring = record          (* type de base *)
                        longueur: integer;
                        chaine : array[1..N] of byte
                 end;
     texte  =  file  of  byte;  (* type  des  fichiers
                                   - la structure des records
                                   est simulée au sein des
                                   procédures Get_typstring
                                   et Put_typstring *)

var file_in: texte;             (* fichier à décoder *)
    file_out: texte;            (* fichier résultat *)
    pos_point:integer;          (* pour la construction du nom
                                   du fichier résultat *)
    filenam1,filenam2:parmtype; (* nom des deux fichiers *)
    str1, str2:typstring;       (* les deux strings
                                   de travail *)
    OK : boolean;               (* si OK=FAUX : le record est
                                   trop court *)


procedure getparm(var s:parmtype);
(* lit le nom du fichier à coder, donné avec l'appel
     du programme dans la ligne de commande *)
var parms: parmtype absolute CSEG:$80; (* MS-DOS dependent !!!!
                                          addresse absolue de la
                                          seconde partie de la
                                          ligne de commande *)
begin
     s:='';
     while ((length(parms)>0) and (parms[1]=' '))
     do delete(parms,1,1);
     while ((length(parms)>0) and (parms[1]<>' '))
     do begin s:=s+parms[1];
              delete(parms,1,1)
        end
end;
```

```
function length_string(str: typstring):integer;
(* correspond la fonction Length_string (voir 6.2) *)
begin
     length_string:=str.longueur
end;


procedure set_length_String(var Str:typstring;
                            Length:integer;
                            var OK:Boolean);
(* correspond à la procédure Set_length_string (voir 6.2) *)
begin
     if (Length<=N)
     then begin Str.longueur:=Length;
                OK:=TRUE
          end
     else OK:=FALSE
end;


procedure get_typstring(var file_in:texte;
                        var str:typstring;
                        var OK:boolean);
(* procédure Get_string (6.2) *)
(* la simulation d'un record se fait par la recherche d'un
     caractère LineFeed (ASCII 10)   *)
var car:byte;

begin
     str.longueur:=0;
     car:=0;
     OK:=TRUE;
     while ((not EOF(file_in)) and (car<>10)
           and (str.longueur<N) and OK)
     do begin
             read(file_in,car);
             if (car=special_car)
             then OK:=FALSE;
             str.longueur:=str.longueur+1;
             str.chaine[str.longueur]:=car
        end;
     (* fin d'un record normalement marquée par CR-LF *)
     if (not EOF(file_in) and (car<>10))
     then OK:=FALSE;
          (* tableau trop petit *)
     if OK then str.longueur:=str.longueur-2
          (* ne pas mettre dans le record
             les caractères CR et LF *)
end;
```

```
procedure put_typstring(var file_out:texte; var str:typstring);
(* procédure Put_string (voir 6.2) *)
var i:integer;
const cr : byte = 13;
      lf : byte = 10;
begin
     for i:=1 to length_string(str) do
          write(file_out,str.chaine[i]);
     write(file_out,cr,lf)  (* fin du record *)
end;


function getcar(str:typstring;pos:integer):byte;
(* fonction Getcar (voir 6.2) *)
begin
     if (pos<=length_string(str))
     then getcar:=str.chaine[pos]
end;


procedure setcar(var str:typstring; pos:integer; car:byte);
(* procedure setcar (voir 6.2) *)
begin
     if (pos<=length_string(str))
     then str.chaine[pos]:=car
end;


procedure delta_decode(var str1, str2: typstring;
                       special_car: byte);
(* procédure réalisant l'opération inverse de celle
     décrite par l'algorithme 6.1 - voir 6.2 *)
var i:integer;
temp:typstring;
begin
     i:=1;
     while ((i<=length_string(str1))
          and (i<=length_string(str2)))
     do begin
        if (getcar(str2,i)=special_car)
        then setcar(str2,i,getcar(str1,i));
        i:=i+1
        end;
     str1:=str2
end;
```

```
(* procédure principale de décodage - voir 6.2 *)
begin
     getparm(filenam1);
     (* pour obtenir le nom du premier fichier *)
     pos_point:=pos('.',filenam1);
     if pos_point=0
     then filenam2:=concat(filenam1,'.und')
     else begin
          filenam2:=copy(filenam1,1,pos_point-1);
          filenam2:=concat(filenam2,'.UNL')
          end;
     (* construction du nom du deuxième fichier =
        nom du premier fichier + extension "UNL" *)
     assign(file_out,filenam2);
     assign(file_in,filenam1);
     reset(file_in);
     rewrite(file_out);
     set_length_string(str1,0,OK);
     (* mise à zéro de Str1 *)
     while (not EOF(file_in) and OK) do
     begin
          get_typstring(file_in,str2,OK);
          if OK
          then begin
                    delta_decode(str1,str2,special_car);
                    put_typstring(file_out,str2)
               end
          else writeln('Je ne peux pas décoder ce fichier !')
     end;
     close(file_in);
     close(file_out)
end.
```

Annexe 6 : Mesures de Compression de Données ARCHIVE (BS2000)

Taille Bloc	Taux Compr. R-L	Temps Trait. R-L	Taux Compr. Huffman	Temps Trait. Huffman	Taux Compr. R-L + Huffman	Temps Trait. R-L + Huffman
214	58.41%	9	119.63%	345	116.82%	299
29067	36.94%	832	44.40%	29304	27.94%	14601
31006	23.25%	758	44.71%	32731	17.46%	10100
31006	33.29%	849	46.96%	33539	24.13%	14116
31006	19.65%	728	43.44%	31436	14.98%	8685
31006	21.69%	745	44.10%	31484	16.60%	9489
31006	16.53%	698	43.21%	31758	13.06%	7496
31006	24.96%	777	45.07%	32162	18.63%	10947
31006	28.68%	808	46.65%	32766	21.06%	12172
31006	15.44%	686	42.39%	32130	12.08%	6854
31006	14.34%	681	42.55%	32430	11.51%	6579
31006	12.41%	663	42.06%	31551	10.01%	5779
31006	11.56%	658	42.40%	31426	9.52%	5216
31006	6.42%	607	28.35%	26414	5.28%	3226
31006	6.18%	605	39.77%	31052	5.13%	3164
31006	1.89%	565	38.00%	29297	1.27%	1272
31006	6.17%	604	39.79%	31619	5.20%	3135
31006	20.86%	737	43.88%	32709	15.82%	9229
31006	19.45%	727	43.89%	31168	14.79%	8642
31006	14.26%	678	42.43%	31549	11.28%	6573
31006	14.75%	680	32.30%	26340	11.11%	6593
31006	23.34%	766	45.08%	31326	17.59%	10129
31006	7.28%	614	40.12%	30701	6.05%	3575
31006	1.94%	567	38.05%	31608	1.36%	1305
31006	1.93%	566	25.59%	25637	1.33%	1288
31006	1.94%	567	38.05%	31600	1.35%	1302
31006	6.97%	610	40.10%	30612	5.96%	3457
31006	10.54%	644	41.26%	30155	8.60%	4981
31006	27.51%	799	45.77%	31902	20.84%	11956
31006	25.58%	786	45.50%	32408	19.38%	11178
31006	30.89%	839	47.05%	32254	23.09%	13329
31006	32.14%	844	46.80%	32439	23.33%	13699
31006	40.94%	946	52.84%	33824	31.31%	17480
29061	58.16%	1009	57.44%	33924	39.29%	21907
31006	56.19%	1065	56.68%	35693	37.57%	22703
31006	56.86%	1077	57.08%	36565	37.95%	23277
31006	57.58%	1087	57.00%	36995	38.70%	23489
31006	56.65%	1068	57.57%	35563	38.94%	23061
31006	44.42%	949	48.80%	32459	31.29%	18307
31006	56.14%	1049	58.42%	35977	39.23%	22992
31006	56.91%	1072	58.60%	36528	39.75%	23391
31006	56.44%	1075	58.64%	32646	39.79%	23397
31006	46.00%	965	56.51%	36392	32.57%	19067
31006	47.09%	983	56.75%	36280	33.75%	19886
31006	44.29%	955	56.41%	36244	31.63%	18463
31006	48.30%	994	56.75%	36427	34.06%	19955
31006	47.06%	986	56.95%	35704	33.63%	19672

Taille Bloc	Taux Compr. R-L	Temps Trait. R-L	Taux Compr. Huffman	Temps Trait. Huffman	Taux Compr. R-L + Huffman	Temps Trait. R-L + Huffman
31006	47.76%	984	56.43%	36205	33.77%	19895
31006	46.83%	977	56.68%	35538	33.35%	19428
31006	46.28%	976	56.37%	35584	32.57%	19188
31006	43.67%	945	56.04%	36216	30.87%	18250
31006	41.43%	925	55.85%	35803	29.13%	17075
31006	47.41%	987	57.62%	36470	33.54%	19488
31006	54.31%	1055	58.70%	36344	38.31%	22545
31006	47.52%	993	56.35%	36303	33.19%	19620
31006	47.61%	989	57.00%	35633	34.13%	19906
31006	49.48%	1001	57.86%	35947	35.42%	20742
31006	43.92%	950	55.57%	36082	30.82%	18261
31006	47.03%	986	57.07%	35728	33.73%	19819
31006	44.70%	957	56.61%	36333	31.70%	18587
31006	45.89%	970	56.90%	36051	32.50%	19174
31006	48.60%	1001	56.79%	36101	33.45%	19709
31006	44.62%	958	55.97%	34812	31.30%	18203
31006	47.31%	983	57.26%	36211	33.35%	19387
31006	50.10%	1019	57.53%	35564	35.66%	20615
31006	48.51%	1004	56.87%	35296	33.95%	19829
31006	50.42%	1013	58.01%	36109	36.25%	20939
16558	49.78%	541	58.82%	19061	36.42%	11146
29072	78.57%	1255	73.74%	37758	66.81%	33299
31005	97.46%	1573	83.65%	43520	82.35%	44038
31005	94.86%	1533	83.60%	43349	80.49%	42586
31005	94.57%	1520	82.23%	42748	79.77%	42794
31005	95.20%	1540	84.60%	43744	80.97%	43010
31005	92.41%	1506	82.69%	43128	79.86%	42611
31005	97.10%	1566	83.51%	43682	82.24%	44133
31005	96.49%	1565	84.23%	43781	82.45%	44016
31005	98.11%	1585	85.02%	43706	84.20%	44992
26877	87.49%	1257	80.45%	36738	74.72%	34395

(Les temps sont donnés en 1/10000 èmes de seconde)