REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI TIZI-OUZOU

FACULTE GENIE ELECTRIQUE ET INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

Mémoire

De fin d'études

En vue d'obtention de diplôme de master en informatique

Option : Conduite de Projets Informatique

Thème

Implémentation et évaluation d'une nouvelle méthode de compression des données basée sur des automates à états finis pour des fichiers textes

Réalisé par :

-M^lle^ NAIT ALI Célia.

-M^lle^ OUDJIDANE Saliha.

Proposé par :

M^r^ S.SADOU

Promotion

2013/2014

Remerciements

Nous remercions le Tout Puissant, qui nous a donné la force et la patience pour l'accomplissement de ce travail.

Nous présentons nos remerciements les plus sincères à notre promoteur Mr S.SADOU, pour avoir été très disponible, pour ses conseils, et pour nous avoir soutenus et encouragés tout le long de l'année.

Nous tenons ensuite à remercier Chacun des membres du jury pour nous avoir fait l'insigne honneur d'accepter d'examiner notre travail.

Merci à nos familles et amis pour leurs soutiens et leurs encouragements.
Nous remercions également toute personne ayant contribué de prés ou de loin à l'aboutissement de cette quête.

Saliha & Célia

Dédicaces

Je dédie ce modeste travail A:

Mes très chères parents adorés.

Ma sœur et mes frères.

Ma grand- mère.

Toute ma famille.

Tous mes amis.

Tous ceux qui m'ont aidé de prés ou de loin dans mes études.

Célia

Dédicaces

Je dédie ce modeste travail A:

Mes très chères parents.

Mes sœurs et mes frères.

Mon neveu Ilyane.

Toute ma famille.

Tous mes précieux amies et amis.

Toute personne que j'estime.

Saliha

Table des matières

Chapitre II : Les approches de compression basées sur les arbres

Liste des figures

Chapitre I :

Chapitre II :

Chapitre III :

Chapitre IV :

Liste des tableaux

Chapitre I :

Chapitre II :

Chapitre III :

Chapitre IV :

Introduction générale

Introduction Générale

L'évolution de l'informatique permise par le génie de l'homme et la machine a sans aucun doute amélioré la communication humaine. Depuis ses débuts, l'informatique ne se contente pas seulement de résoudre des problèmes mais plutôt de les résoudre avec un maximum d'efficacité. Cette discipline connait une véritable révolution malgré les contraintes technologiques non négligeables rencontrées tout au long de son évolution.

Le stockage et la transmission des données d'une façon économique ont toujours constitué un problème de taille à cause de la limite des capacités des supports de stockage et du débit des lignes de transmission. Les premières solutions étaient basées sur l'extension des unités de stockage et l'augmentation des vitesses de transmissions. Hélas, ces solutions n'apportent qu'une réponse à court terme au problème de stockage et toutefois, il convient de noter qu'elles sont coûteuses et peu efficaces. Heureusement, la compression de données est considérée comme solution qui peut être employée pour alléger une portion du problème de stockage et de transfert des données.

La nouvelle méthode de compression étudiée ici est basée sur un nouveau code proposé, ce dernier consiste à compresser des données de type texte, elle est sans perte car elle assure une copie exacte de l'original lors de la décompression.

Et pour bien mener notre travail, nous le répartissons sur quatre chapitres :

- **le premier chapitre** est une introduction générale à la compression de données, dans laquelle nous présentons les définitions essentielles et les différents concepts fondamentaux de la compression de données.
- **Le second chapitre** à pour objectif de présenter quelques approches de compression de données basées sur les arbres, ainsi que quelques algorithmes et techniques de construction de ces codes.

- **Le troisième chapitre** illustre d'une part, la technique de compression utilisée, en effet il met l'accent sur les fondements et les concepts de cette nouvelle méthode de compression ; et d'autre part, nous décrivons d'une façon détaillée l'analyse et la conception de la démarche adoptée où nous présentons les différents modules et algorithmes à développer qui sont nécessaires à la mise en œuvre de notre application.
- les résultats d'implémentation et de différents tests d'évaluation de notre méthode, précédée par une présentation des différentes fonctions principales et outils nécessaires pour la réalisation de notre application, feront l'objet du **dernier chapitre.**

Enfin, nous clôturons par une conclusion générale où nous montrerons les apports de ce travail.

Chapitre I

Généralités sur la compression des données

1. Introduction :

Les données informatiques occupent un espace disque non négligeable, de même lorsqu'on souhaite transmettre des données sur le réseau internet, et la bande passante allouée pour le transfert est très limitée en termes de débit et de capacité de transfert, pour ces raisons la compression de données est considérée comme solution qui peut être employée pour alléger une portion de problème de stockage pour faire en sorte que les données occupent le minimum d'espace pour une vitesse de restriction acceptable.

2. La théorie de l'information : [1]

C'est la théorie mathématique traitant des bases théoriques de la transmission et du traitement de l'information. La théorie de l'information est une approche statistique dont les premiers rudiments ont été proposés par HARLEY en 1928, mais dont les fondements n'ont vraiment été établis qu'en 1948 par le mathématicien CLAUDE SHANNON dans son livre intitulé « théorie mathématique de la communication ».

Cette théorie s'intéresse a la mesure de la quantité d'information, à la représentation de cette information, dite aussi codage, ainsi qu'aux systèmes de communication qui la transmettent et la traitent. Ce codage peut ainsi se référer à la conversion du son et d'image en signaux électromagnétiques.

3. Les éléments de la théorie de l'information :

3.1.Quantité d'information : [2]

Pour une source S tel que S=(s, P), la quantité d'information fournit par l'occurrence s tel que s appartient à S est :

$$H(S) = Log_2\left[\frac{1}{P(S)}\right] = -Log_2[p(S)]. \quad Sh$$

Ou log2 désigne logarithme binaire et P(S) est la probabilité d'apparition de symboles s tel que :

$$P(s) = \frac{Fréquence}{taille\ en\ nombre\ de\ symboles}$$

3.2.Entropie d'un texte: [3]

En général, l'entropie mesure le degré de désordre dans un système, en théorie de l'information, il indique le poids d'information mathématique que porte un message, et donc sa compressibilité théorique. L'entropie (H) d'un message est l'information moyenne contenue par chaque symbole, elle est donnée par la relation suivante :

$$\mathbf{H(X) = \sum_{i=1}^{n} p(i)\, log2 \left(\frac{1}{p(i)}\right) = -\sum_{i=1}^{n} p(i)\, log2\, p(i)}$$

Remarque :

L'entropie est maximale H_{max} (x) si la probabilité d'apparition des symboles est égale à $\frac{1}{n}$

3.2.1. Utilité de l'entropie :

D'un strict point de vue quantitatif, on ne peut que constater que la plupart des langues comprenant beaucoup moins de mots que les possibilités offertes par l'alphabet, ainsi les codages traditionnels, qui représentent ou cherchent à représenter toutes les combinaisons théoriques, se révèlent extrêmes simpliste et peu performants. Le codage ASCII sur 7 bits, par exemple, fournit plus de 4 000 milliards de possibilités pour coder les mots de 8 lettres, alors que les dictionnaires de la langue française comptent moins de 3000.

En tenant compte des caractéristiques d'entropie des données à traiter, il est donc possible de définir des codages plus performants. Les travaux de Shannon et ses collaborateurs ont conduit à développer des codages fondés sur la fréquence d'apparition d'une information.

3.3. La redondance : [15]

La compression de données se fonde sur la détection et l'élimination des données redondantes. Sachant que la donnée informatique (texte, image, vidéo ….etc.) est représentée sur une combinaison de 256 bits. Alors chaque flot de donnée supérieur présente une redondance.

La redondance **R** d'une source caractérise la différence qu'il existe entre la quantité d'information que transportée cette source et la quantité d'information que cette source transporterait si tous les symboles étaient équiprobables et indépendants. Elle est mesurée par : $\mathbf{R = 1-(H(x) / H_{max}(x))}$

4. Le codage et le code :

4.1.Définition :

Permet d'établir une correspondance qui permet sans ambigüité de passer d'une représentation d'une information à une autre représentation sous forme binaire de la même information suivant un ensemble de règles précises.

4.2.Propriété d'un code :

Pour avoir un codage correct et bon il faut qu'il vérifie certaines propriétés :

- Tous les mots du code doivent être distingués.
- Le décodage ne donne lieu à aucune ambigüité.
- Il doit être préfixé (aucun mot de code n'est un sous mot initial d'un autre), cette propriété permet d'assurer un décodage unique.

4.3.Types de codes : [4] [5]

Les différents types de code peuvent être représentés selon leur taille sous les formes suivantes :

- **Code à longueur fixe (FLC)** : (Fixed Length Codes) est un code qui associe à tous les messages de la source un nombre constant de bits.
- **Code à longueur variable(VLC) :** (Variable Lenght Codes) est un code qui associe les symboles de la source à un nombre variable de bits.

- les codes à longueur variable sont souvent choisit pour la compression des données, par ce que toute opération d'enregistrement des données assigne des codes plus court pour les symboles plus fréquents, et des codes plus long pour les symboles plus rares.

Exemple *:*

Considérons le code a longueur variable suivant (0, 100, 101, 110, 111) avec des longueurs de codes (1, 3, 3, 3,3) pour l'alphabet (A, B, C, D, E) et la chaine source BAAAAAAAC avec des fréquences pour chaque symbole (7, 1, 1, 0, 0).

La moyenne en nombre de bits est :

$$L=\frac{1*7+3*1+3*1}{9}= 1.4 \ bits/symbole$$

C'est presque un enregistrement avec la moitié de bits comparé avec 3bits/symbole si on utilise un code de longueur fixe à 3 bits.

Le plus court code est attribué au plus fréquent symbole dans la chaine source.

4.4. Déchiffrabilité d'un code : [4] [5]

Un code C sur un vocabulaire V est dit uniquement déchiffrable (on dit parfois non ambigu) si et seulement si, pour tout x = x1 … $x_n \in$ V +, il existe au plus une séquence c = c1 … $c_m \in$ C+ telle que :

c1 …c_m = x1 …x_n

Propriété : Un code C sur un vocabulaire V est uniquement déchiffrable si et seulement si pour toutes séquences c = c1 … c_n et d = d1 … d_m de C+ :

c = d→ (n = m et 1 <= i <= n; $c_i = d_i$).

Exemples :

- C = (0, 01, 001) n'est pas uniquement déchiffrable.
- C = (01, 10) est uniquement déchiffrable.
- C = (0, 10, 110) est uniquement déchiffrable.

Théorème 1 (Kraft) :

Il existe un code uniquement déchiffrable sur un vocabulaire V dont les mots {c_1 … c_n} sont de longueur {l_1… l_n} si et seulement si :

$$\sum_{i=0}^{n} \frac{1}{|V|^{li}} <= 1$$

4.5.Propriété de préfixe : [4] [5]

On dit qu'un code C sur un vocabulaire V a la propriété du préfixe (on dit parfois qu'il est instantané, ou irréductible) si et seulement si pour tout couple de mots de code distincts (c_1, c_2), c_2 n'est pas un préfixe de c_1.

Exemple :

(a = 101000), (b = 01), (c = 1010) b n'est pas un préfixe de a mais c est un préfixe de a.

Grâce à la propriété du préfixe, on peut déchiffrer les mots d'un tel code dès la fin de la réception du mot ce qui n'est pas toujours le cas pour les codes uniquement déchiffrables : par exemple, si V = 0, 01, 11 et si on reçoit le message m = 001111111111 …, il faut attendre l'occurrence suivante d'un 0 pour pouvoir déchiffrer le second mot (0 ou 01 ?).

Propriétés :

- Tout code possède la propriété de préfixe est uniquement déchiffrable.
- Tout code dont tous les mots sont de la même longueur possède la propriété de préfixe.

4.6. Classification de codage : [15]

Il existe 3 types de codage :

1. **Codage de canal :** est utilisé pour le cryptage de données à transmettre pour des raisons de sécurité d'information.
2. **Codage correcteur d'erreurs :** est utilisé pour la correction des erreurs de transmission.
3. **Codage de la source :** est utilisé pour la compression des données.

5. La compression et la décompression : [14]

5.1. Définition :

La compression de données ou codage de source est l'opération informatique qui consiste à transformer une suite de bits A en une suite de bits B plus courte, en utilisant un algorithme particulier.

Il s'agit d'une opération de codage, C'est-à-dire changer la représentation de l'information, dans le but de rendre la représentation compressée plus courte que la représentation originale.

La décompression est l'opération inverse de la compression.

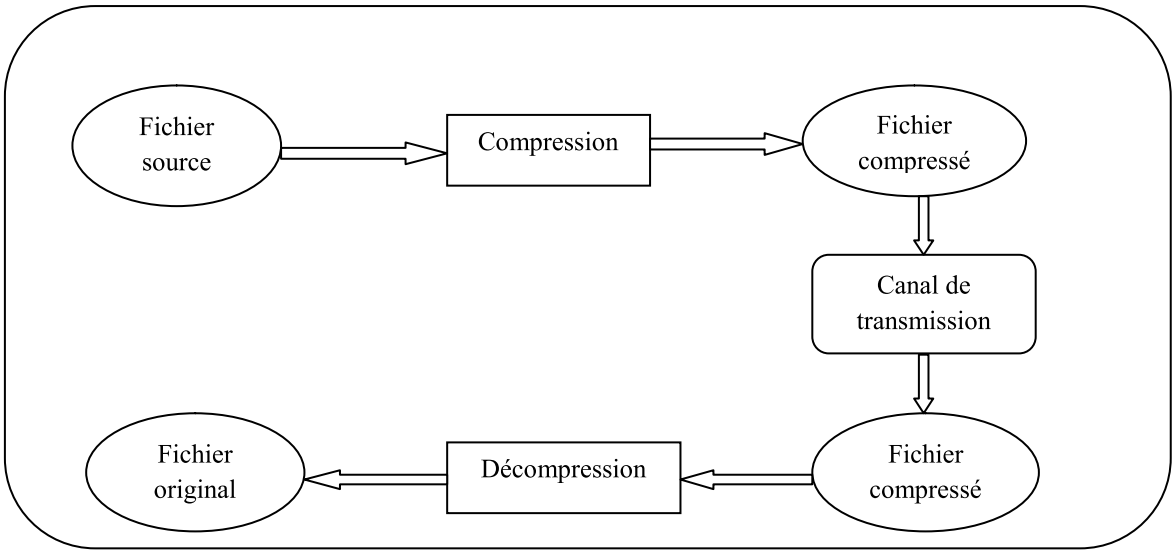


Figure 1. Schéma représentatif de la compression et de la décompression

5.2. Principe de la compression :

Pour réduire la place qu'occupe un fichier sur la mémoire, nous utiliserons des logiciels de compression.ces derniers utilisent des algorithmes, qui sont une suite d'opérations mathématiques, propre au type de données contenues dans le fichier afin de coder l'information initiale qui après la compression puis la décompression, sera retrouvé soit à l'identique soit légèrement modifiée.

5.3.Objectif de la compression : [6]

De nos jours, la puissance des processeurs augmente plus vite que les capacités de stockage, et énormément plus vite que la bande passante des réseaux (car cela imposerait d'énormes changements dans les infrastructures de télécommunication).

Il y a donc un déséquilibre entre le volume des données qu'il est possible de traiter, de stocker, et de transférer.

Par conséquent, il faut donc réduire la taille des données. Pour cela, il faut exploiter la puissance des processeurs, pour pallier aux insuffisances des capacités de stockage en mémoire et des vitesses de transmission sur les réseaux.

6. La différence entre le codage et la compression :

Le codage c'est l'envoi des données de l'émetteur vers le récepteur par le canal sous forme des signaux (des codes), mais la compression de données c'est de réduire la taille de la donnée qu'on veut envoyer à l'émetteur pour réduire le temps de transmission.

7. Classification des algorithmes de compression :

7.1. Compression physique/logique : [6]

On considère généralement la compression comme un algorithme capable de comprimer des données dans un minimum de place (compression physique), mais on peut également adopter une autre approche et considérer qu'en premier lieu un algorithme de compression a pour but de recoder les données dans une représentation différente plus compacte contenant la même information (compression logique).

La distinction entre compression physique et logique se base sur la façon dont les données sont compressées ou plus précisément comment est-ce que les données sont réarrangées.

- **La compression physique :** est exécutée exclusivement sur les informations contenues dans les données. Cette méthode produit typiquement des résultats incompréhensibles qui apparemment n'ont aucun sens. Le résultat d'un bloc de données compressées est plus petit que l'original car l'algorithme de compression physique a retiré la redondance qui existait entre les données elles-mêmes.
- **La compression logique :** est accomplie à travers le processus de substitution logique qui consiste à remplacer un symbole alphabétique, numérique ou binaire en un autre. Changer "United State of America" en "USA" est un bon exemple de substitution logique car "USA" est dérivé directement de l'information contenue dans la chaîne "United State of America" et garde la même signification. La substitution logique ne fonctionne qu'au niveau du caractère ou plus haut et est basée exclusivement sur l'information contenue à l'intérieur même des données.

7.2. Compression symétrique / asymétrique : [6]

Dans le cas de la compression symétrique, la même méthode est utilisée pour compresser et décompresser l'information, il faut donc la même quantité de travail pour chacune de ces opérations. C'est ce type de compression qui est généralement utilisée dans les transmissions de données.

La compression asymétrique demande plus de travail pour l'une des deux opérations, la plupart des algorithmes requiert plus de temps de traitement pour la compression que pour la décompression. Des algorithmes plus rapides en compression qu'en décompression peuvent être nécessaires lorsque l'on archive des données auxquelles on accède peu souvent (pour des raisons de sécurité par exemple).

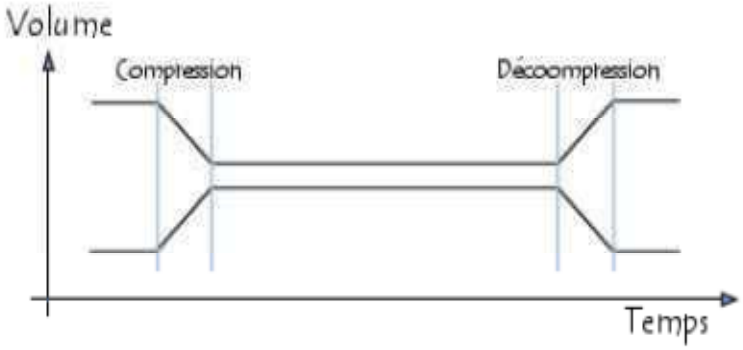


Figure 2. Compression de type symétrique.

7.3.La compression statistique / numérique : [6]

On peut encore distinguer les algorithmes qui travaillent au niveau statistique et ceux qui opèrent au niveau numérique.

Pour les premiers, la valeur des motifs ne compte pas. Ce sont les probabilités qui comptent, et le résultat est inchangé par substitution des motifs tandis que pour les seconds, les valeurs des motifs influent sur la compression (par exemple JPEG), et les substitutions sont interdites.

Enfin le critère de classification le plus pertinent est basé sur la perte des données.

7.4.Encodage adaptif, semi adaptif et non adaptif : [7]

Certains algorithmes de compression sont basés sur des dictionnaires spécifiques à un type de données : ce sont des encodeurs **non adaptifs**. Les occurrences de lettres dans un fichier texte par exemple dépendent de la langue dans laquelle celui-ci est écrit.

Un encodeur **adaptif** s'adapte aux données qu'il va devoir compresser, il ne part pas avec un dictionnaire déjà préparé pour un type de données.

Enfin un encodeur **semi-adaptif** construira celui-ci en fonction des données à compresser : il construit le dictionnaire en parcourant le fichier, puis compresse ce dernier.

7.5.La compression sans perte (Non destructive): [6]

Les algorithmes de compression sans perte (connu aussi sous le nom de non destructible, réversible, ou conservative) sont des techniques permettant une reconstitution exacte de l'information après le cycle de compression / décompression.

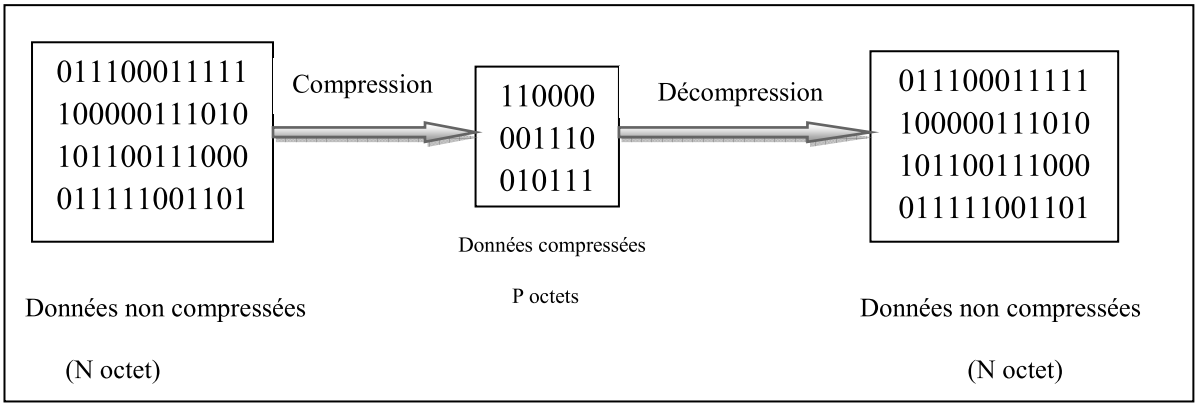


Figure 3. Principe de la compression sans perte.

7.5.1. Méthodes de compression des données sans perte :

7.5.1.1. Compression par substitution :

7.5.1.1.1. Codage RLE (Run Length Encoding): [9]

1) Principe :

Le codage RLE (Run-Length-Encoding), ou « codage par plage » consiste à réduire la taille d'un texte en remplaçant des suites de caractères identiques en couple de valeurs (caractère remplacé, nombre d'occurrences).

Exemple :

« AAACCCCC » peut être représenté par « 3*[A], 5*[C] ».

2) L'encodage sans séparateur :

Le codage le plus simple consiste à ne pas utiliser de séparateur pour représenter les plages de valeurs.

Exemple :

« AAAA » sera remplacé par « A4 ».

Moins de caractères sont utilisés, ce qui permet un meilleur taux de compression que la version avec séparateur, mais le texte en entrée ne doit contenir aucun chiffre. On peut cependant coder des données numériques en remplaçant les chiffres par des lettres (ex : 0 par a, 1 par b….).

Cet algorithme peut dans certains cas produire en sortie un texte plus grand que le texte initial, par exemple :

« A » remplacé par « 1A » → Le gain est négatif pour une seule occurrence.

« AA » est remplacé par « 2A ». → Le gain est nul.

« AAA » est remplacé par « 3A ». →Le gain est positif.

Par convention, il a été décidé de ne coder que les plages de longueur >2, et copier tel quel les caractères dans le cas échéant ;

3) L'encodage avec séparateur :

Il est nécessaire d'utiliser un séparateur pour encoder des textes contenant des chiffres. Si « # » est choisi, la séquence « AAAAA» sera alors coder « #5#A ».Le nombre d'octets nécessaire est alors augmenté de 2, on encodera alors uniquement les plages de 5 symboles. Ex: « 5*A »sera codé « #5#A ».

Par la suite pour simplifier, le séparateur sera toujours écrit #, et |#|représentera le nombre d'occurrence consécutives de #.

Il existe un cas particulier, lorsque le caractère # est rencontré, celui-ci doit être encodé d'une certaine façon pour que le décodeur puisse l'interpréter correctement. Le principe de doublement de caractère normal, par exemple « #20## ».

Remarque:

- Ce type de codage est utilisé dans la majorité des algorithmes existants dans le but de réduire le nombre de caractères en supprimant les redondances.
- La version sans séparateur sera utilisée lorsque les données à encoder ne contiennent aucune donnée numérique, dans tous les autres cas la version avec séparateur sera préférée(en choisissant de préférence un caractère peu ou pas présent dans le texte).

7.5.1.2.Méthodes à base statistique :

1) Le codage de Shannon-Fano : [2] [10]

Ce codage est basé sur l'idée de répartir les symboles en deux groupes de valeur à peu prés équivalente, cette valeur étant la somme, dans chaque groupe, les probabilités d'apparition des symboles qu'il contient. Le groupe de gauche est appelé 0, celui de droite 1(ce choix est arbitraire). Les groupes sont à nouveau subdivisés et nommés 0 ou 1jusqu'à ce que la subdivision contienne plus qu'un symbole. L'arbre binaire obtenu est formé de segment ou branches et de feuilles contiennent un caractère simple. Pour déterminer le code numérique d'un caractère donné, il faut partir du sommet de l'arbre et suivre les branches jusqu'à atteindre la feuille qui le présente. Les caractères les plus fréquents se trouvent le plus prés du sommet et requièrent donc moins de bits dans leurs transcriptions compressée.

2) Codage de Huffman : [6]

L'algorithme de codage de Huffman est un algorithme de compression des données basé sur les fréquences d'apparition des caractères de la source. Les caractères les plus fréquents de la source sont codés par des mots courts et les caractères les moins fréquents par des mots plus longs. Cet algorithme permet de générer un code préfixé de taille variable.
Cela est plus efficace que d'utiliser une longueur fixe pour chaque caractère, comme le code ASCII qui utilise un octet par caractère. Cet algorithme a été inventé en 1952 et est encore utilisé de nos jours.

Remarque :

Les deux algorithmes précédents (Shannon et Huffman) se basent sur la création des arbres binaires alors, ils seront étudiés en détail dans le prochain chapitre.

7.5.1.3. Méthodes à base de dictionnaire :

7.5.1.3.1. L'algorithme LZW (Lempel Ziv Welch) : [20]

1) Description de l'algorithme :

LZW (Lempel Ziv Welch) est un algorithme de compression de données sans perte. Il s'agit d'une amélioration de l'algorithme LZ78 inventé par Abraham Lempel et Jacob Ziv en 1978. LZW fut créé en 1984 par Terry Welch d'où son nom.

Cet algorithme de compression consiste à réaliser la construction d'un dictionnaire. Les caractères inferieurs à 256 sont initialement présents dans le dictionnaire. A mesure que l'algorithme examine le texte, il ajoute de nouvelles chaines de caractères dans le dictionnaire.

- Ce principe forme la base de la compression LZW (Lepel Ziv Welch). La méthode est semblable à la méthode RLE, mais appliqués à des suites d'octets.
- Cet algorithme réduit la taille des chaines de caractères (c'est-à-dire les mots) récurrents.

2) Algorithme de compression : [15]

```
w= NUL ;
tant que (lecture d'un caractère c) faire
    si (wc existe dans le dictionnaire) alors
        w = wc ;
        sinon ajouter wc au dictionnaire ;
        écrire le code de w ;
        w = c ;
    fin si
fin tant que
écrire le code de w ;
```

Exemple :

Le tableau suivant illustre le fonctionnement de l'algorithme LZW.

- On a la chaine suivante : « **TOBEORNOTTOBEORTOBEORNOT ».**
- Après l'exécution de l'algorithme de compression sur la chaine précédente on aura le résultat suivant :

C	W	wc	Sortie	dictionnaire
T		T		
O	T	TO	T	TO=<256>
B	O	OB	O	OB=<257>
E	B	BE	B	BE=<258>
O	E	EO	E	EO=<259>
R	O	OR	O	OR=<260>
N	R	RN	R	RN=<261>
O	N	NO	N	NO=<262>
T	O	OT	O	OT=<263>
T	T	TT	T	TT=<264>
O	T	TO		
B	TO	TOB	<256>	TOB=<265>
E	B	BE		
O	BE	BEO	<258>	BEO=<266>
R	O	OR		
T	OR	ORT	<260>	ORT=<267>
O	T	TO		
B	TO	TOB		
E	TOB	TOBE	<265>	TOBE=<268>
O	E	EO		
R	EO	EOR	<259>	EOR=<269>
N	R	RN		
O	RN	RNO	<261>	RNO=<270>
T	O	OT		
	TO		<263>	

Tableau 1. Exemple de l'algorithme LZW

Apres la compression, nous obtenons une séquence de code de 9 bits sur la sortie : **TOBEORNOT.**

Remarque :

Les programmes de compactage les plus performants combinent la puissance des algorithmes de type LZW et l'efficacité des algorithmes statistiques pour compresser les motifs redondants. La combinaison est simple :

- Aux algorithmes de type dictionnaire de détecter les redondances dans les fichiers et de constituer les dictionnaires de motifs redondants.
- Aux algorithmes statistiques de trouver les codages les plus concis (précis) pour les termes contenus dans les dictionnaires.

3) Algorithme de décompression :

Lors de la décompression, l'algorithme reconstruit le dictionnaire dans le sens inverse, ce dernier n'a donc pas besoin d'être stocké.

4) Utilité et caractéristique de l'algorithme :

Cette méthode est peu efficace pour les images mais donne de bons résultats pour les textes et les données informatiques en générale.

Remarque :

Le taux de compression des algorithmes sans perte est en moyenne de l'ordre de 40% pour des données de type texte. Par contre, ce taux est insuffisant pour les données de type multimédia. Il faut donc utiliser un nouveau type de compression pour résoudre se problème, la compression avec perte.

7.6.Compression avec perte : [6]

Son principe est basé sur l'étude précise de l'œil et de l'oreille humaine. Les signaux audio et vidéo contiennent une part importante de données que l'œil et l'oreille ne peuvent pas percevoir et une part importante de données redondantes.

Les objectifs de la compression avec pertes sont d'éliminer les données non pertinentes pour ne transmettre que ce qui est perceptible et, comme pour la compression sans perte, d'éliminer l'information redondante.

Ce type de compression engendre une dégradation indiscernable à l'œil (ou à l'oreille) ou suffisamment faible, en contrepartie d'un taux de compression très élevé.

Il existe des algorithmes de compression consacrés à des usages particuliers, dont en voici 3 :

- Compression du son (Audio MPEG, ADCPM ...).
- Compression des images fixes (JPEG,...).
- Compression des images animées (MPEG, ...).

8. Les types de données à compresser :

8.1. Le texte :

De manière générale, toute information structurée en ASCII ou composée de caractères alphanumériques. Dans un traitement de texte et en PAO (publication assistée par ordinateur), partie principale d'un document, par opposition aux titres, tableaux, figures, notes de bas de page et autre éléments.

8.2. Le son :

Un son peut être défini comme une vibration généré mécaniquement, transmise généralement par aire sous forme d'ondes qui aboutissent au tympan de l'oreille, avant d'être interprété par le cerveau c'est donc un signal que l'on peut représenter sous forme d'une courbe mathématique indiquant l'intensité en fonction du temps qui s'appelle un signale analogique ce signale doit encor être numérisé pour pouvoir être exploiter par l'ordinateur pour cela, il est échantillonné, c'et à dire découpé dans le temps, par une carte son.

8.3.L'image : [8]

C'est une représentation plane d'un objet tridimensionnelle, perçu par l'œil humain, ou plus généralement un capteur dont le fonctionnement est semblable (exemple d'objets : une scène, un portrait, une échographie, une observation astronomique…). Elle peut être décrite

sous la forme d'une fonction I(x, y) de brillance analogique continue définie dans un domaine borné, tel que x et y sont les coordonnés spatiales d'un point de l'image et I est une fonction d'intensité lumineuse ou de couleur.

8.3.1. La définition :

Pour une image numérisée, on utilise une autre notion qui est la définition, laquelle peut s'exprimer par plusieurs valeurs, soit le nombre de points par ligne, soit le nombre de lignes, soit le produit des deux.

8.3.2. La numérisation :

La numérisation d'une image est la conversion de celle-ci de son état analogique en une image numérique représentée par une matrice bidirectionnelle de valeur numérique f(x, y).

Ou x, y coordonnées cartésiennes d'un point d'image.

8.3.2.1.Mode de numérisation :

- **Mode de niveau gris :**

Dans ce mode la valeur du rouge (R), du vert (V) et du bleu (B) est la même, donc codé sur n=8 bits, nous pouvons donc définir 256 niveaux de gris, allant du niveau 0 (noir) au niveau 255 (blanc).

- **Mode couleur RGB :**

RGB (Red-Green-Blue) ce triplet permet de définir une couleur : chaque couleur peut prendre une valeur de 0 à 255, indiquant son intensité.par exemple un triplet (255, 0, 0) représente la couleur rouge foncé. De même (0, 0, 0) représente le noir, et (255, 255, 255) représente le blanc. C'est ce qu'on appelle la synthèse additive (rouge+vert+blanc) à laquelle notre œil est sensible, $(2^8)^3$ couleurs, soit pratiquement ce que l'œil humaine peut voir. Cette échelle additive s'imagine facilement en ajoutant des couleurs à la couleur noire. Donc plus on ajoute de couleurs (R, V, B), plus on se rapproche du blanc. Elle est utilisée pour des écrans.

Elle s'oppose à l'échelle soustractive, où l'on soustrait des couleurs à la couleur blanc (CMYK : Cyan, MAgeta, Yellow, blak ; soit CMJN en français). Cette quadrichromie est le

principe utilisé par les imprimantes couleurs, la photographie. On trouve également d'autres modèles, comme le YUV (luminance et chrominance), utilisé pour les téléviseurs (et le format JPEG), et HSV (ou HSB) : HUE, saturation, Brightness.

8.4.La vidéo :

Technique de transformation d'images animées en signaux électriques (signaux vidéo), destiné à permettre leur diffusion ou leur enregistrement. Les premiers pas en terme de compression ont été fait a partir du JPEG et on donné naissance au M-PEGEB (Moving JPEG), qui consiste à compresser chaque image au format JPEG. Ceci a amené à coder par JPEG la première image, et à coder les différences entre chaque pair d'images successives par JPEG également.

9. Performance de la compression : [8]

Pour mesurer l'efficacité et les performances d'une compression il faut se baser sur les critères suivants :

9.1. Le taux de compression:

- Le degré de compression :

Le degré de compression des données se mesure par la comparaison de la taille des données à compresser à la taille des données initiales, et nommé aussi le quotient de compression donnée par la formule suivante :

$$\textit{Quotient de compression}=\frac{\textit{Taille des données initiales}}{\textit{Taille des données compressées}}$$

9.2. Le gain de compression :

Le gain de compression représenté en %, c'est l'espace dont on a bénéficie après une opération de compression telle que plus le gain est important, plus la compression est efficace, il est donné par la formule suivante :

Gain de compression=100*taux de compression

9.3. Le taux d'information :

Qui peut servir à noter la différence entre la qualité et la quantité d'information.

9.4. Mesure de fidélité :

C'est la distorsion introduite lors des deux processus de compression/décompression par rapport à un fichier source.

9.5. La vitesse de compression :

C'est le temps mis par l'opération compression/décompression sur le flux de donnée. Il y a plusieurs facteurs qui permettent de juger la rapidité d'un compresseur.

9.6. Mesure de perte d'information :

C'est un critère d'évaluation des performances d'un compresseur/décompresseur de donnée qui ne peut restituer, les données originales.

Conclusion :

Dans ce chapitre nous avons présenté les différentes notions de base de la compression et son intérêt, ou nous avons vu des différentes méthodes pour compresser tout types de données (texte, son, image, vidéo), et enfin nous avons exposé quelques mesures de l'évaluation de performance d'une compression.

Dans le chapitre suivant nous allons traiter les différentes approches de compression basées sur les arbres.

Chapitre II

Les approches de compression basées sur les arbres

1. Introduction

Dans cette partie on s'intéresse aux méthodes de compression de données et les algorithmes correspondants. Ce sont des méthodes de codage de la source d'information.

La compression ne doit pas conduire à des distorsions saisissables et doit être réalisée avec taux de compression les plus élevés possibles. Dans la suite on présente quelques méthodes de compression pour les textes qui utilisent des arbres binaires, et des automates à états finis.

2. Définition d'un arbre binaire : [15]

En informatique, un arbre binaire est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé *racine*. Dans un arbre binaire, chaque élément possède au plus deux éléments fils au niveau inférieur, habituellement appelés *gauche* et *droit*. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé *père*.

Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé *feuille*. Le nombre de niveaux total, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé *hauteur* de l'arbre.

Le niveau d'un nœud est appelé *profondeur*.

3. Les approches de compression basées sur les arbres :

3.1.Code préfixe et arbre binaire :

3.1.1. Code préfixe :

Un code préfixe (ou code instantané) est un code ayant la particularité de ne posséder aucun mot ayant pour préfixe un autre mot. Autrement dit, aucun mot fini d'un code préfixe ne peut se prolonger pour donner un autre mot.

Exemple :

On considère les mots des deux codes suivant: (0, 10, 010, 101) et (0, 10, 110, 111). Le premier mot de code (0, 10, 010, 101) n'est pas un code préfixe car le symbole codé par 10 est le préfixe du symbole codé par 101. Par contre le mot de code (0, 10, 110, 111) est un code préfixe, car aucun symbole est le préfixe d'un autre symbole. C'est une propriété souvent recherchée pour les codes à longueur variable, afin de pouvoir les décoder lorsque plusieurs mots sont concaténés les uns aux autres sans qu'il soit nécessaire d'utiliser des séparateurs (les séparateurs rendent préfixes des codes non-préfixes). Ce sont des codes non-ambigus. Les codes à taille fixe sont tous des codes préfixes par exemple (01, 11, 01) est un code préfixe.

3.1.2. Construction de l'arbre binaire : [11]

Soit S : texte à coder

A : alphabet

f (c) : fréquence du caractère c dans S

On représente un codage préfixe par un arbre binaire où chaque feuille correspond à un caractère de l'alphabet, et chaque branche est étiquetée par la valeur 0 ou 1 (un bit).

- à chaque caractère *c* correspond un code Φ (c) (suite de bits)
- aucun code n'est le préfixe d'un autre
- le codage est optimal si : $\sum_{c\in A} f\,(c) \times |\Phi(c)|$ est minimal (taille du texte après codage)

a. le codage :

Soit un texte à coder S.

- Calcul de l'alphabet A (et des fréquences)
- Calcul des codes des caractères de l'alphabet
- Parcours du texte en produisant à la place de chaque caractère son code (parcours des feuilles vers la racine lorsqu'on utilise l'arbre de codage)

Exemple : soit un texte à coder :

S : « dacadbaac »

- calcul de l'alphabet : A= {a, b, c, d}
- construction de l'arbre binaire et calcul des codes des caractères de l'alphabet :

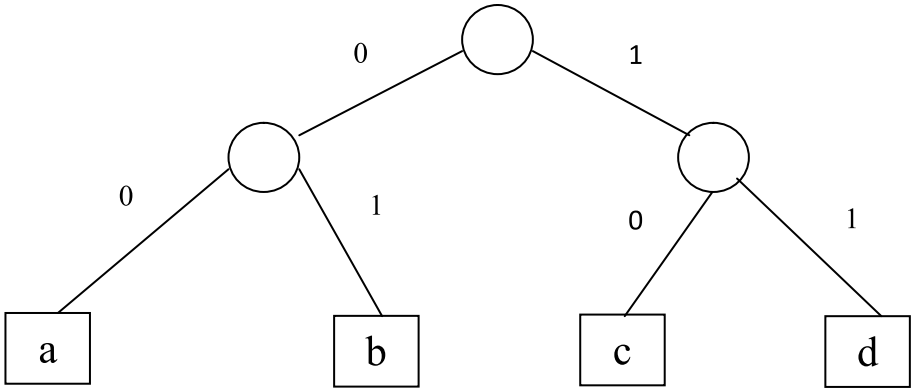


Figure 4. Exemple de code.

caractère	code
A	00
B	01
C	10
D	11

- Parcours du texte en produisant à la place de chaque caractère son code (parcours des feuilles vers la racine lorsqu'on utilise l'arbre de codage) :

« dacadbaac » ⟶ 11-00-10-00-11-01-00-00-10 (taille=18)

b. Le Décodage :

Codes + texte codé ⟶ texte original

- Il n'y a pas d'ambigüité dans un codage préfixe, puisqu'aucun code n'est le préfixe d'un autre
- analyse du texte codé en recherchant les codes (parcours de la racine vers les feuilles dans l'arbre de codage)
- La représentation des codes sous la forme d'un arbre de codage facilite l'analyse.

c. Le Codage préfixe optimal :

L'idée est de construire un arbre où les caractères les plus fréquents ont des codes plus courts.

L'arbre de codage est complet

Pour l'alphabet A, les fréquences f et l'arbre T : $B(T) = \sum_{c \in A} f\,(c) \times \mathrm{pt}(c)$ est minimale ($p_t(c)$ représente la profondeur de la feuille d'étiquette c)

Reprenons l'exemple précédent :

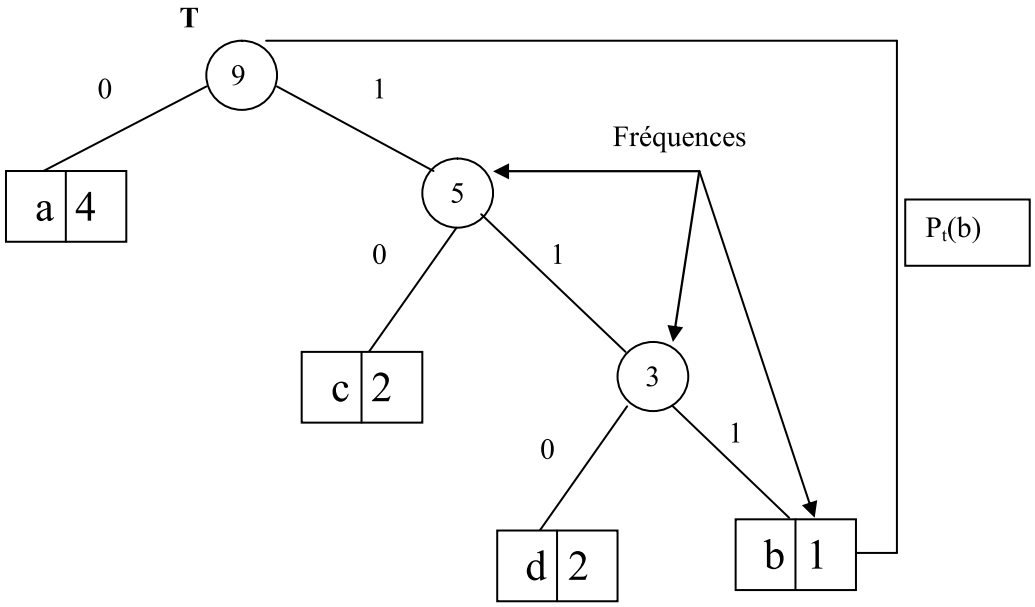


Figure 5. Arbre de codage optimal étiqueté par les caractères et les fréquences

caractère	code
A	0
B	111
C	10
D	110

« dacadbaac » ⟶ 110-0-10-0-110-111-0-0-10 (taille=**17**)

3.1.3. Algorithmes et techniques de construction du code préfixe :

3.1.3.1. Algorithme de Huffman : [6] [12]

Le codage de Huffman est un algorithme de compression de données sans perte basé sur les fréquences d'apparition des caractères apparaissant dans la source, il a été développé par David Albert Huffman en 1952. C'est un codage entropique de type VLC (codage à longueur variable) qui consiste à coder les caractères qui apparaissent souvent dans un texte par un code binaire court, et ceux qui apparaissent plus rarement par un code plus long.

3.1.3.1.1. Algorithme de compression :

1. A chaque symbole de la source, on associe sa fréquence. Chaque symbole de fréquence non nulle représente un arbre binaire à un élément.
2. Classer chaque arbre binaire par valeur croissante de racine.
3. Enlever de la liste les 2 arbres binaires de plus faible racine. Rattacher ces 2 arbres à une nouvelle racine qui vaut la somme des racines des 2 arbres précédents.
4. Recommencer en 2) jusqu'à ce qu'il ne reste plus qu'un arbre binaire unique.

a. La table de fréquence:

Afin de construire cette table, il suffit simplement de dénombrer le nombre d'occurrences de chaque symbole S puis de calculer la fréquence f_s de chacun d'entre eux grâce à la formule suivante :

$$f_s = \frac{nombre\ d'occurences\ des\ symboles}{nombre\ de\ symboles}$$

Ensuite en trie le tableau en fonction de la fréquence d'appariation (par ordre croissant).

b. Construction de l'arbre :

L'arbre binaire de Huffman c'est la structure de données qui nous permet d'attribuer à chaque symbole une représentation binaire optimale. Afin de construire l'arbre, on utilise la table de fréquences construite précédemment qu'on appelle T et on applique l'algorithme suivant :

Données :

T : table de fréquence.

Q : une file d'attente de nœuds de l'arbre binaire, chaque feuille est étiquetée avec un symbole et son nombre d'occurrences.

Chaque nœud interne est étiqueté avec la somme des occurrences des feuilles de sa sous arborescence.

O : une fonction qui a chaque nœud de l'arbre associe une valeur. Si le nœud est une feuille alors O renvoie le nombre d'occurrence du symbole, autrement O renvoie la somme des occurrences des feuilles de la sous arborescence de nœud.

Résultat :

A : Arbre binaire.

Begin

Initialisation de Q tel que Q contient les feuilles représentant les symboles de la table T

Tant que (Q non vide) **faire**

Créer un nouveau nœud z dans A tel que :

Gauche(z)=x=extraire-min(Q)

Droite(z)=y=extraire-min(Q)

O(z)=O(x)+O(y)

Insérer (z, Q)

Fin

End.

La file d'attente Q est utilisée pour placer les nœuds correspondants au couple (symbole, Nombre d'occurrence) pour tous les symboles. Puis on extrait de la file d'attente les deux nœuds ayant la valeur minimale, en créant un nouveau nœud dans l'arbre ayant pour fils les deux sélectionnés, on rajoute le nœud nouveau nœud à la file d'attente, et on réitère jusqu'à ce que la file soit vide.

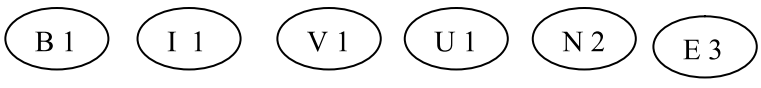
3.1.3.1.2. La Décompression :

Il nous faut cette fois réaliser l'opération inverse. A partir d'une liste de bits, il faut recomposer par un parcours dans l'arbre le mot initial.

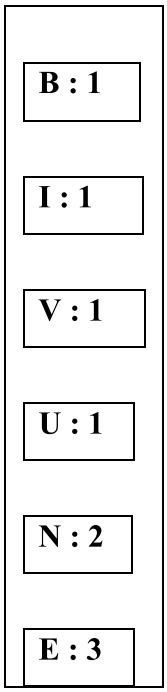
3.1.3.1.3. Exemple d'application : Nous voulons coder le mot : « bienvenue »

Symbole	**b**	**I**	**V**	**u**	**n**	**e**
Fréquence	**1**	**1**	**1**	**1**	**2**	**3**

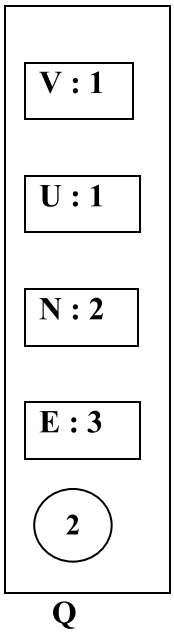
Etape 1 : La création des nœuds.

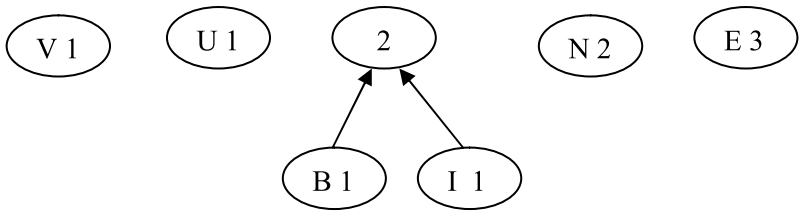


Initialisation de la file d'attente :

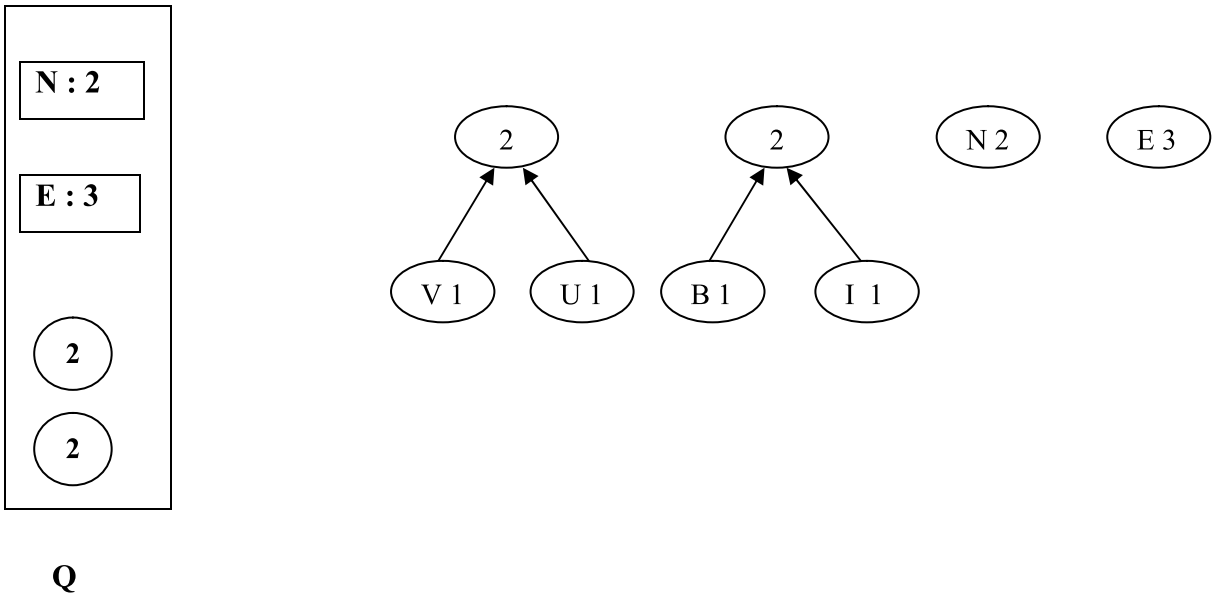


Etape 2 :

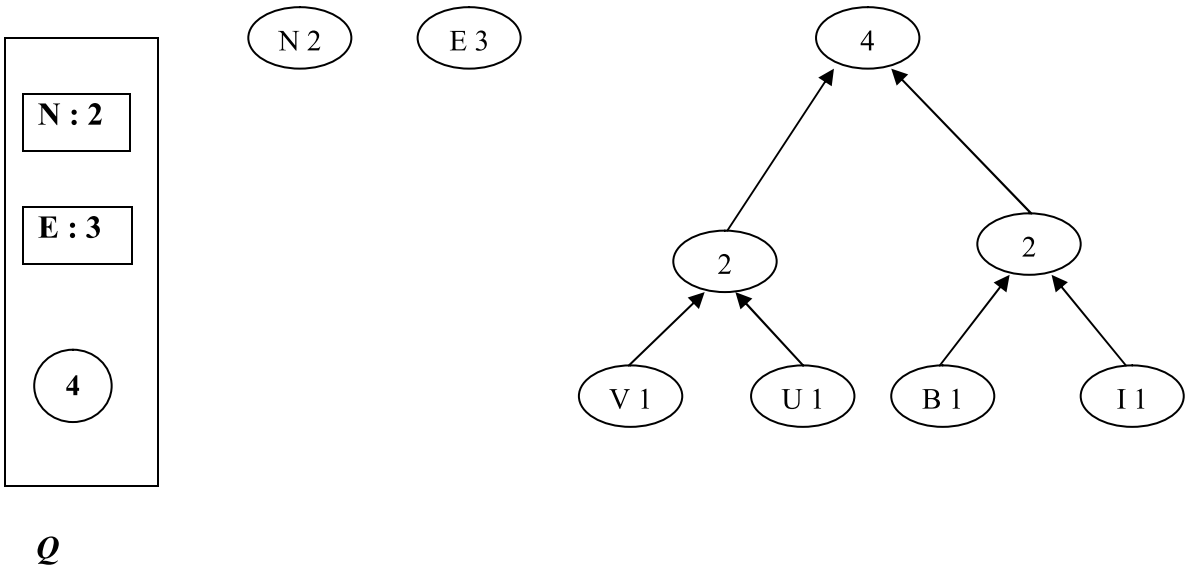




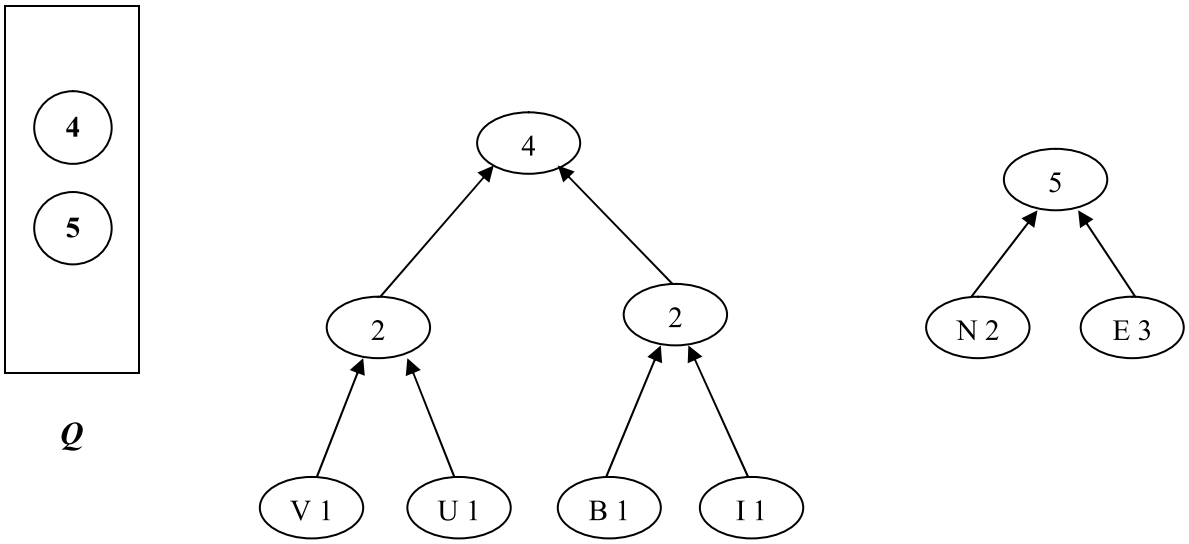
Etape 3:



Etape 4 :



Etape 5 :



Etape 6 :

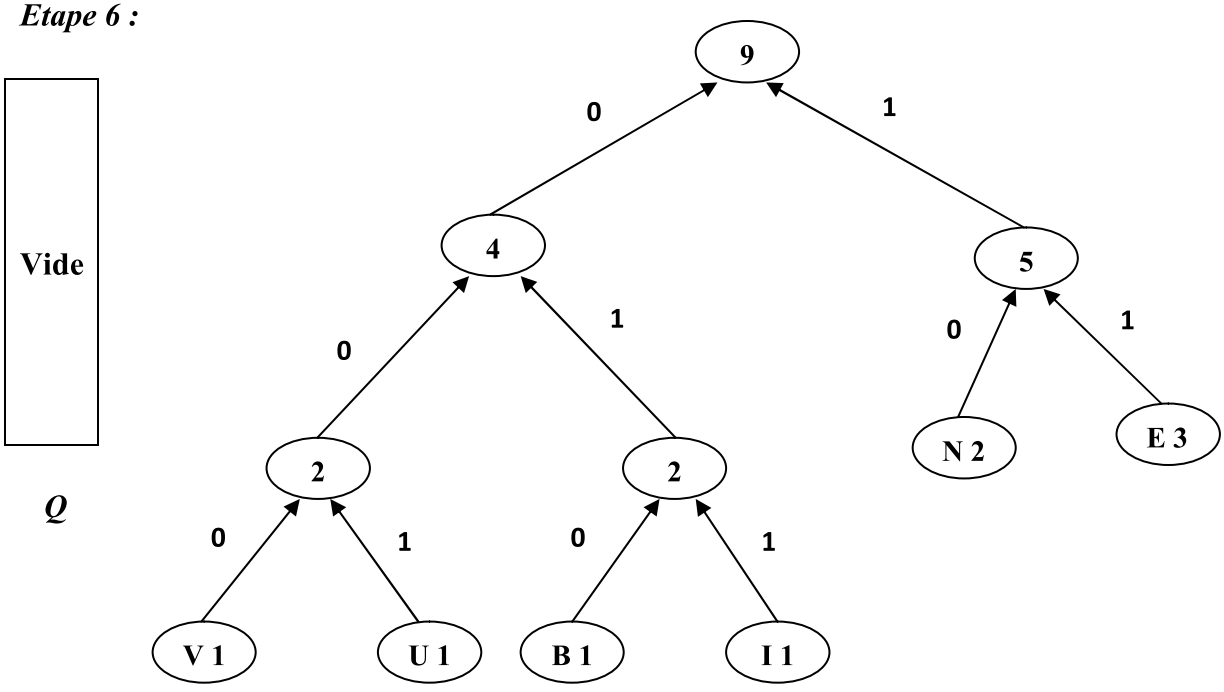


Figure 6. Arbre de Huffman.

L'arborescence de Huffman finalement construite, on étiquette les arcs de la façon suivante :

- Les arcs reliant un nœud à son fils gauche sont étiquetés par '0'.
- Les arcs reliant un nœud à son fils droit sont étiquetés par '1'.

De cette manière, chaque feuille représentant un symbole peut être redéfinie par un nombre binaire correspondant au chemin entre la racine et la feuille de l'arborescence. Ainsi les symboles les plus utilisés ont une représentation binaire moins importante (en termes de taille) que les symboles les moins utilisés. Ceci permet de représenter chaque symbole de façon optimale et permet de réaliser une compression des données efficace.

Caractère	Représentation en bit	Taille en bit	Gain en bit
e	***11***	**2**	**6**
n	***10***	**2**	**6**
i	***011***	**3**	**5**
b	***010***	**3**	**5**
u	***001***	**3**	**5**
v	***000***	**3**	**5**

Tableau 2. Tableau de correspondance pour l'algorithme de Huffman.

Le mot « bienvenue » codé devient : 0100111110000111000111, Ce qui fait 22 bits au lieu de 72 en ASCII.

3.1.3.1.4. Performances sur l'exemple :

L'intérêt de la compression est de pouvoir réduire au maximum la taille des informations originales, et cette mesure peut être effectuée par le taux de compression qui est défini parr la formule suivante :

$$\mathbf{P=1-}\frac{taille\ compréssée}{taille\ originale}$$

- Taille originale :

  Nombre de caractère de la source, et chaque caractère dans le code ASCII est sur 8 bits, **Taille originale =9*8**=72 bits.
- Taille compressée : c'est la somme de la taille de la représentation binaire de chaque symbole,

  Taille compressée =22 bits
- **Le taux de compression :**

$$\mathbf{P = 1 - \frac{taille\ compréssée}{taille\ originale} = 1 - \frac{22}{72} = 0.6944}$$

Le taux de compression dans cet exemple est proche de 70 %, ce qui est relativement très bon.

3.1.3.2. Le codage de Shannon-Fano : [13]

3.1.3.2.1. Principe de l'algorithme :

L'idée est de répartir les symboles en deux groupes de valeur à peu près équivalente, cette valeur étant la somme, dans chaque groupe, des probabilités d'apparition des symboles qu'il contient. Le groupe de gauche est appelé 0, celui de droite 1 (ce choix est arbitraire).

Les groupes sont à nouveau subdivisés et nommés 0 ou 1 jusqu'à ce que la subdivision ne contienne plus qu'un symbole. L'arbre binaire ainsi obtenu est formé de segments ou branches et de feuilles. Chaque branche représente un bit d'informations (0 ou 1), Chaque feuille contient un caractère simple. Pour déterminer le code numérique d'un caractère donné, il faut partir du sommet de l'arbre et suivre les branches jusqu'à atteindre la feuille qui le représente. Les caractères les plus fréquents se trouvent le plus près du sommet et requièrent donc moins de bits dans leurs transcriptions compressées.

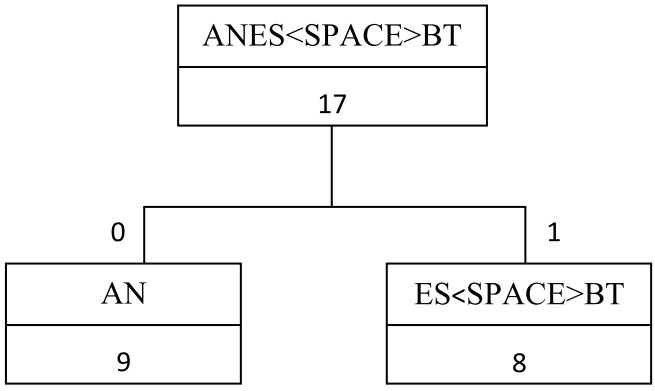
3.1.3.2.2. Exemple de codage de Shannon-Fano :

Supposons que nous voulions transmettre le message suivant : "BANANES ET ANANAS". Il compte 17 caractères. Nous allons compter les occurrences des différents caractères de l'alphabet utilisés et les classer par fréquences décroissantes. Puis nous allons regrouper les caractères en deux groupes dont les fréquences d'apparition sont aussi proches que possibles puis diviser chacun de ces groupes de la même façon jusqu'à parvenir à chacune des fréquences de départ (il est équivalent de travailler avec le nombre d'occurrences ou les fréquences).

- **Première subdivision selon Shannon-Fano :**

La subdivision définit deux symboles (qui sont en fait des groupes de symboles), "AN" et ES<space>BT. Cette subdivision est ainsi faite pour que les fréquences des deux subdivisions soient aussi équilibrées que possible.

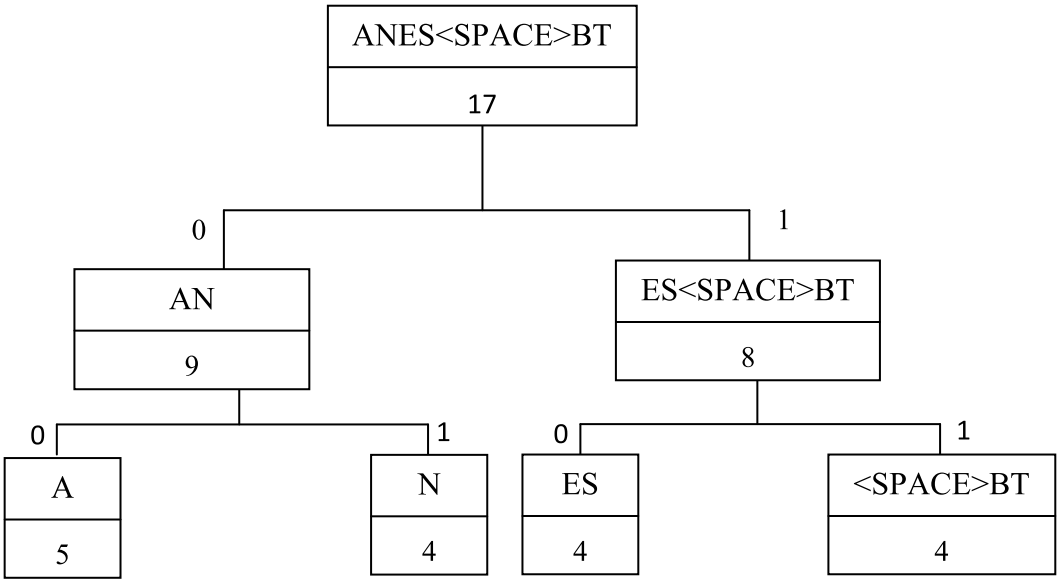
Caractère	**fréquence**	
A	5	9
N	4	
E	2	8
S	2	
<SPACE>	2	
B	1	
T	1	



- **Deuxième subdivision selon Shannon-Fano :**

La deuxième étape va isoler les symboles A et N, et définir une nouvelle subdivision des symboles restants.

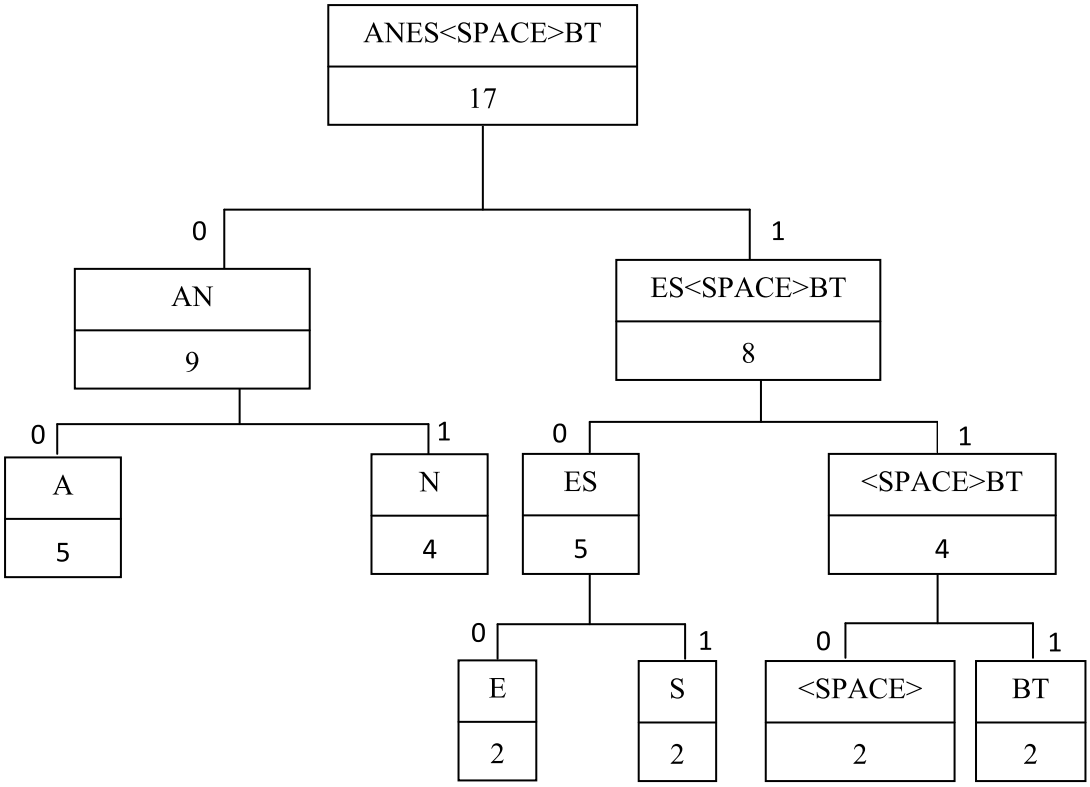
Caractère	fréquence	
A	5	
N	4	
E	2	4
S	2	
<SPACE>	2	4
B	1	
T	1	



- **Troisième subdivision selon Shannon-Fano :**

La troisième étape va isoler les symboles E, S et <SPACE>, et définir une nouvelle subdivision des symboles restants.

Caractère	fréquence
A	5
N	4
E	2
S	2
<SPACE>	2
B	1
T	1



- **Quatrième subdivision selon Shannon-Fano (partition finale) :**

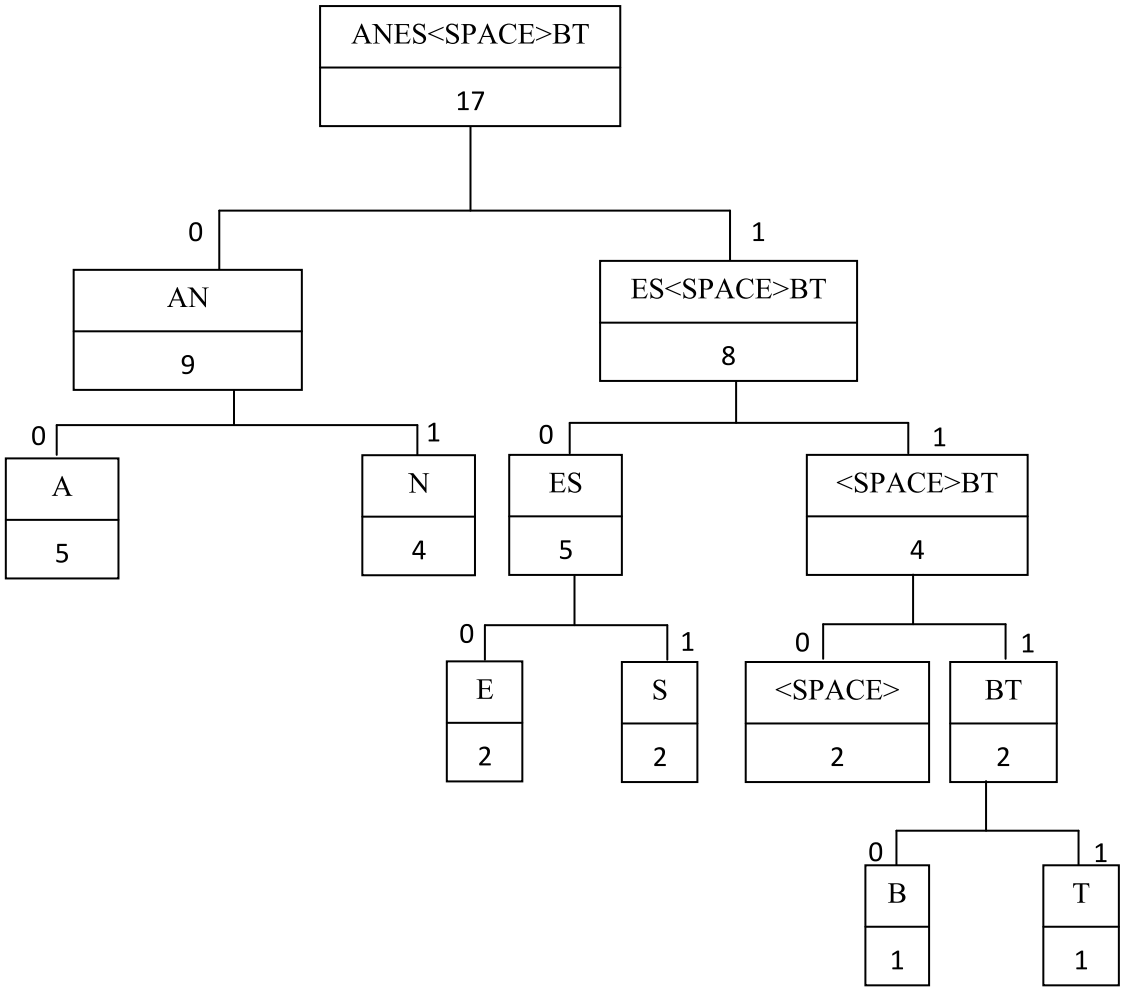


Figure 7. Arbre de Shannon-Fano

Pour connaître le code associé à chaque lettre, on parcourt l'arbre final de haut en bas, et l'on obtient :

Caractère	Code
A	00
N	01
E	100
S	101
<SPACE>	110
B	1110
T	1111

Tableau 3.Tableau de correspondance pour l'algorithme de Shannon-Fano.

3.2.Codage par automate à états finis :

Les automates à nombre fini d'états sont utilisés dans de nombreux domaines tel que le traitement du langage naturel, la compilation, etc. Dans l'objectif de réduire la mémoire utilisée par des automates, une recherche de structures répétitives a été menée.

3.2.1. Définitions :

a. Un alphabet :

Un **alphabet** est un ensemble, en général supposé fini et non vide. Ses éléments sont des **lettres**.

b. Un mot :

Un **mot** sur un alphabet **A** est une suite finie d'éléments de **A**. Un mot est noté par la juxtaposition de ses lettres. Ainsi, on écrit $\mathbf{u=a_1\ldots a_n}$ au lieu de $\mathbf{u= (a_1,\ldots, a_n)}$. La **longueur** d'un mot est le nombre d'éléments qui le composent. La suite vide, mot de longueur 0, souvent noté**ε**, est appelée le **mot vide**. L'ensemble des mots sur A est noté A^*. La **concaténation** de deux mots $\mathbf{u=a_1\ldots a_n}$ et $\mathbf{v=b_1\ldots b_n}$ est le mot $\mathbf{uv=a_1\ldots a_nb_1\ldots b_n}$ obtenu par juxtaposition. En particulier, **uε=εu=u**. La concaténation est associative.

c. Automate à états finis : [17]

Un automate à états fini est la représentation d'un texte sous forme d'un arbre dont les nœuds représentent les états et les arcs, annotés par des éléments de l'alphabet, décrivent les transitions.

Exemple : construction d'un automate qui représente les mots suivants :

« Soif Soir »

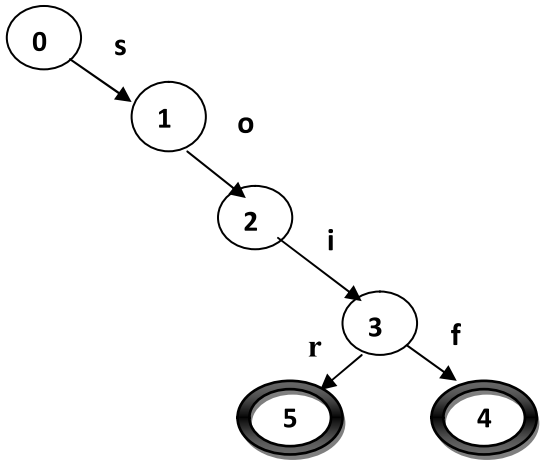


Figure 8. Exemple de codage par automate à états finis.

d. Définition formelle d'un automate : [17]

Un automate à états fini est un quintuplé A = (S; E;T; S0; F) avec :

- S un ensemble fini d'états,
- E un alphabet,
- T une fonction de transition T : S x E → S,
- S0 ⊆ S l'ensemble des états initiaux,
- F ⊆ S l'ensemble des états terminaux.

3.2.2. Ensemble des successeurs et prédécesseurs d'un état :

Les ensembles de successeurs et prédécesseurs d'un état p∈S sont toujours calculés dans l'automate A comme suit :

Succ (p) = {s ∈ S: ∃a∈E: T (p, a) =s}

Succ* (p) = {s ∈ S: ∃w∈E*: T (p, w) =s}

Pred (p) = {s ∈ S: ∃a∈E: T (p, a) =p}

Pred*(p) = {s ∈ S: ∃w∈E*: T (p, w) =p}

Avec:

Succ(p)/ Pred (p) : Ensemble des successeurs/ Prédécesseurs immédiats de P.

Succ* (p)/ Pred*(p) : Ensemble de tous les successeurs/prédécesseurs de P, incluant l'état P.

3.2.3. Un sous-automate : [19]

Les sous-automates sont des sous-structures de l'automate qui possèdent un seul point d'entrée et un seul point de sortie, ils peuvent ainsi être extraits ou remplacés par une transition.

Etant donné les structures que nous souhaitons rechercher, nous avons retenu la définition suivante d'un sous-automate comme suit :

Soit A=<E, S, T, S_i, S_f> un automate acyclique à nombre fini d'états, A'=<E, S', T', S'_i, S'_f> est un sous-automate de A si et seulement si :

S'⊂ S, {S'$_I$, S'$_f$}⊂ S,

T':
- **S' x E→S'**
- **∀ (s, a) ∈S'\ {S'$_i$, S'$_f$} x E : si T(s, a) est défini alors T'(s, a)=T(s, a) sinon T'(s, a) est indéfini.**
- **∀ a E, si T (S'$_i$, a) ∈ S' alors T' (s, a)=T(s, a) sinon T'(s, a)est indéfini**
- **∀ a E, si T' (S'$_f$, a) est indéfini.**

∀ s ∈ S', s ∈Succ*(s'$_i$) et s ∈ Pred*(s'$_f$).

∀ s ∈ S'/ {s'$_i$, s'$_f$}:Succ(s) ⊂ S' et Pred(s) ⊂ S'.

3.2.4. Représentation et stockage de l'automate :

Un automate est représenté en mémoire par la liste de ses transitions sortantes et qu'à chaque transition sont associées 3 informations :

- Un indicateur d'appartenance à l'état courant (un booléen mis à 0 ou à 1 selon que la transition partage ou pas le même état source que celle qui la précède).
- Une étiquette,
- L'adresse de son état d'arrivée.

3.2.5. Minimisation des automates :

Il est connu que les automates représentant des dictionnaires contiennent de nombreuses transitions en série ou en parallèle.

3.2.5.1. Détection de sous automates séries-parallèles : [18]

La méthode s'exécute en plusieurs passes. L'idée conductrice est de remplacer itérativement les transitions parallèles et les transitions séries par une unique transition qui référence un sous automate. La recherche considère d'abord les transitions parallèles, puis les transitions séries, etc. Dans la figure dessous, l'automate initial contient plusieurs transitions en parallèle, par exemple les états 2 et 7 sont reliés par trois transitions. La première passe nous permet d'aplatir directement toutes ces transitions pour obtenir un automate dénué de parallèle. La seconde passe

concatène toutes les séries en une unique transition et donne l'automate A3, en aplatissant de nouveau les parallèles on obtient l'automate A4 et en aplatissant ensuite les séries on obtient l'automate A5. L'algorithme se termine donc sur l'automate A5 qui ne présente plus aucune série ni aucun parallèle. Cet automate reconnaît le même dictionnaire que l'automate A1, mais ses transitions se réfèrent soit à des lettres, soit à des sous-automates de A1.

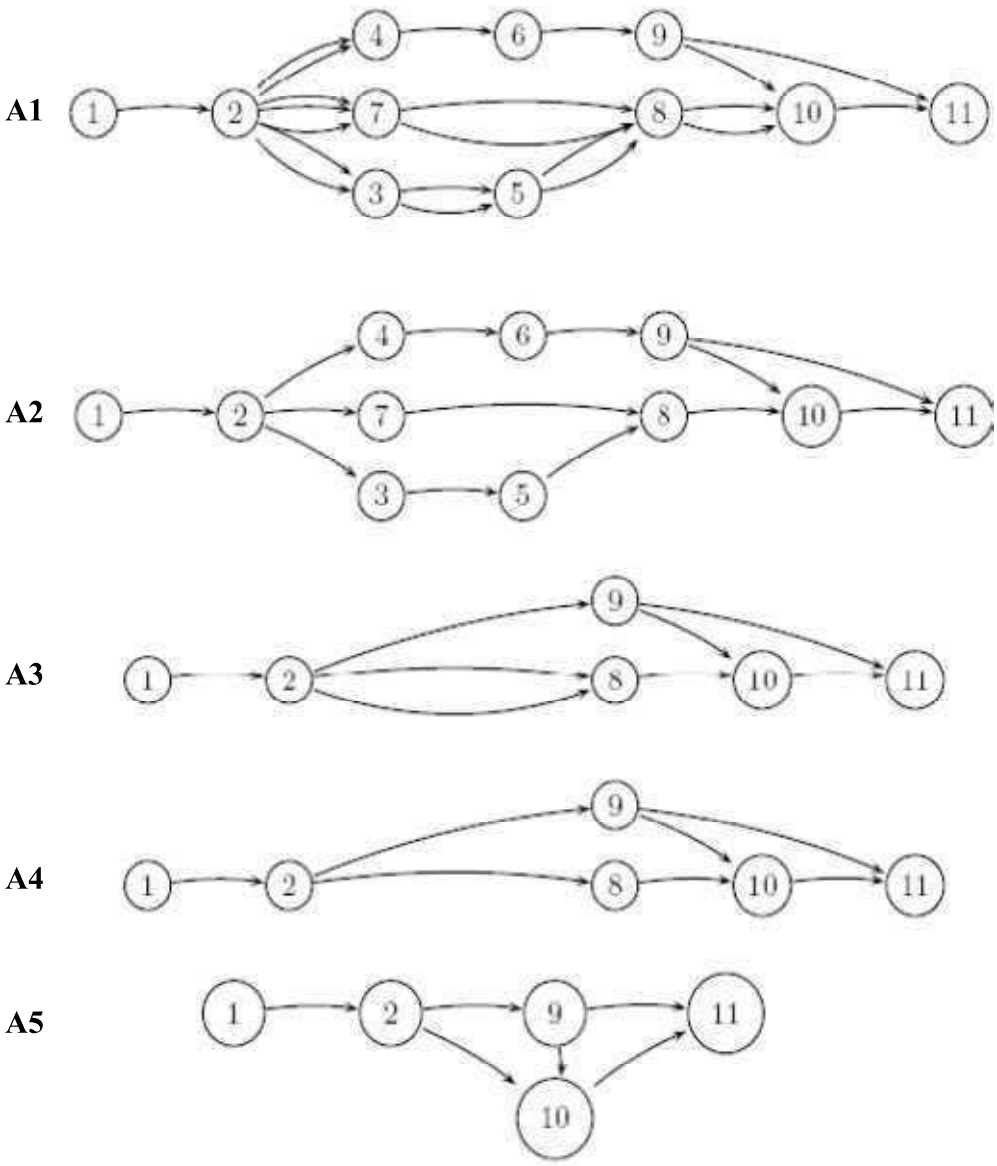


Figure 9. Recherche de sous automates séries-parallèles

3.2.6. Optimisation de l'occupation mémoire d'un automate :

Les automates sont souvent stockés dans des fichiers textes. Dans le but de réduire l'espace mémoire nécessaire, on a codé chaque donnée du même type sur un nombre de bits minimal, mais constant. Cette méthode permet, d'une part de stocker les automates à moindre cout et, d'autre part, de conserver un accès efficace aux données. Le codage réservé à chaque donnée est le suivant :

1. **Codage du booléen :** 1 bits.
2. **Codage des étiquettes :** le nombre de bits nécessaire pour coder l'alphabet, noté Taille $_{alphabet}$: **Taille $_{alphabet}$ = [$\log_2$ (| ∑' |)]** ou ∑' est l'union de l'alphabet initial ∑ et des nouveaux caractères alloués aux séries et parallèles détectés et factorisés.
3. **Codage de l'adresse de l'état d'arrivée :** le nombre de bits nécessaire pour coder des adresses, noté Taille adresse : **Taille $_{adresse}$= [$\log_2$ (valMax+1)]** ou valMax correspond à l'adresse de l'état but de valeur maximale.

Dans l'exemple précédent l'automate initial est composé de 11 états et 21 transitions, après l'optimisation de l'automate, et élimination des sous-automates série et parallèles nous avons obtenu un automate final composant de 6 transitions, et 5 états.

4. Conclusion :

Dans ce chapitre nous avons présenté les différentes approches de compression basées sur les arbres, et qui utilisent le codage préfixe. Et parmi ces méthodes on trouve les algorithmes de Huffman, et de Shannon-Fano, la mise en œuvre de ces deux algorithmes est différente, parce que le codage de Shannon-Fano est récursif à chaque étape il faut découper la table de fréquence en deux, tandis que l'algorithme de Huffman réitère en ajoutant des éléments à la table. De point de vue efficacité l'algorithme de Shannon donne de meilleurs résultats quand la dispersion de fréquences est importante.

Et pour corriger les inconvénients majeurs de l'algorithme de Huffman c'est-à-dire de lire le texte entièrement avant de lancer la compression, une version adaptative à été proposée, et cette évolution est d'utiliser des automates à états finis, pour éviter le problème d'espace mémoire.

Dans le chapitre qui suit nous allons voir la méthode adaptée pour la réalisation de notre application.

Chapitre III

Analyse & conception

1. Introduction

Dans ce qui suit nous allons faire en premier lieu l'analyse de la nouvelle méthode proposée pour la compression de données qui est basée sur les automates d'états finis déterministes, ensuite présenter l'architecture générale de notre application ainsi que ses différents modules en expliquant le processus de codage et de décodage. Ce système doit fournir en plus des fonctionnalités de compression/décompression d'autres fonctions telles que le calcul de temps, le taux de compression et éventuellement les tailles des fichiers.

Enfin nous allons présenter les différents algorithmes nécessaires à la mise en œuvre de notre application.

2. Présentation de la méthode proposée :

Cette méthode est classée parmi les méthodes de compression sans perte, elle est basée sur la création d'un automate à états finis déterministe après la lecture du fichier désiré mot par mot, et la compression se fait par transformation de cet automate par plusieurs étapes en structure moins compliquée pour la stocker en mémoire.

2.1. Principe de la méthode proposée :

2.1.1. La compression : est la phase qui consiste à réduire la taille du fichier source.

Cette phase est réalisée comme suit :

1. Lecture du fichier et extraction des mots contenus dans ce dernier,
2. Construire l'automate d'états finis déterministe,
3. Transformer cet automate en matrice,
4. Puis la matrice en arbre linéaire,
5. Construire une nouvelle matrice à partir de l'arbre,
6. Compression de la nouvelle matrice,
7. Sauvegarder le résultat.

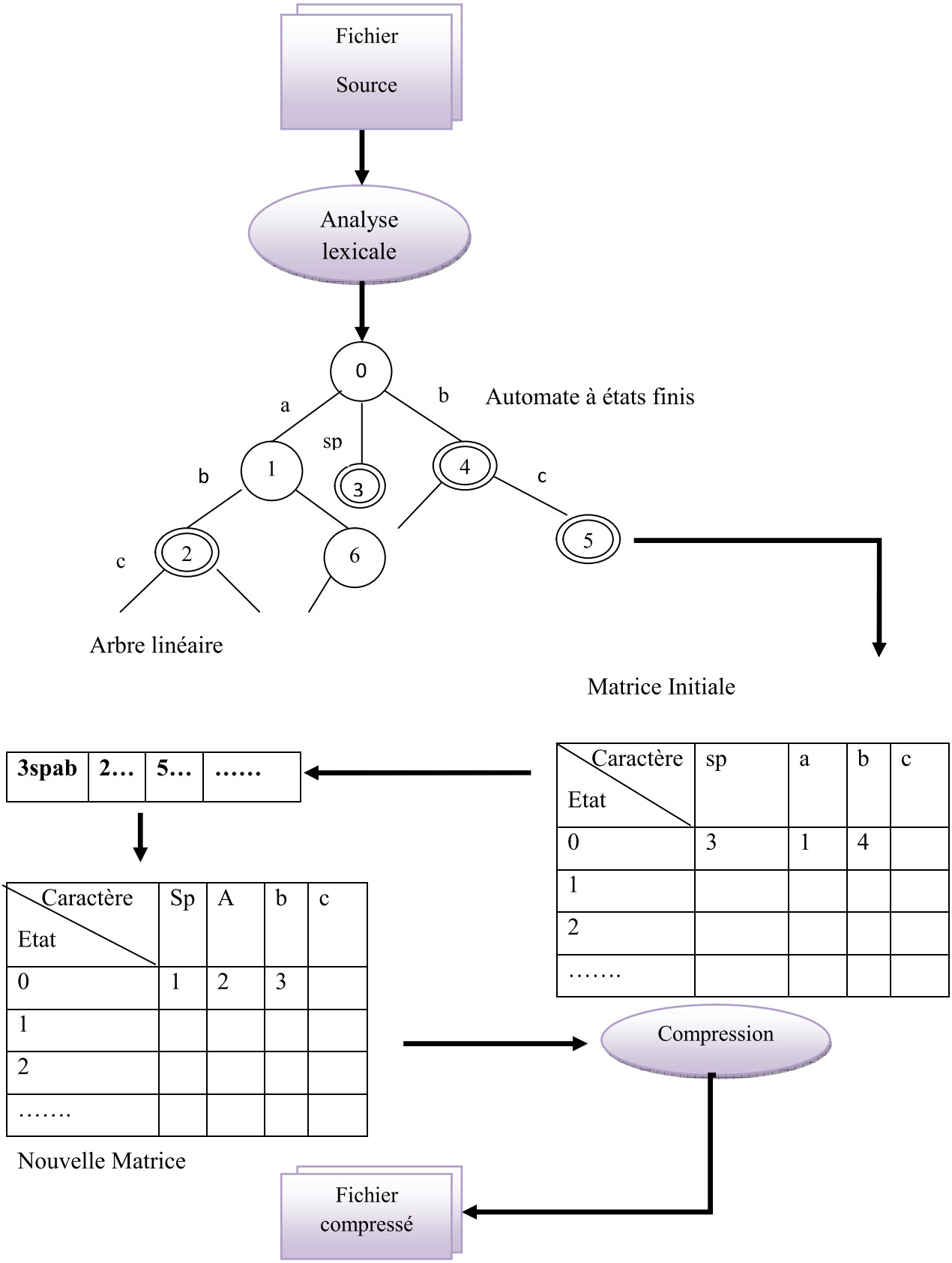


Figure 10. Schéma général de la compression.

2.1.2. La décompression : Cette phase consiste à restaurer le fichier source à partir du fichier compressé.

Cette phase est réalisée comme suit :

1. Récupérer l'arbre linéaire ainsi la nouvelle matrice ;
2. Créer une table final à l aide de l'arbre linéaire contenant les états finaux comme indices, et affecter 1 à tous les états finaux et 0 aux autres.
3. Créer une autre table mot qui va contenir tout les mots récupérés après le parcours de la nouvelle matrice.
4. Récupérer le texte source à l'aide de la table mot et l'arbre linéaire, en suivant l'ordre des états finaux.

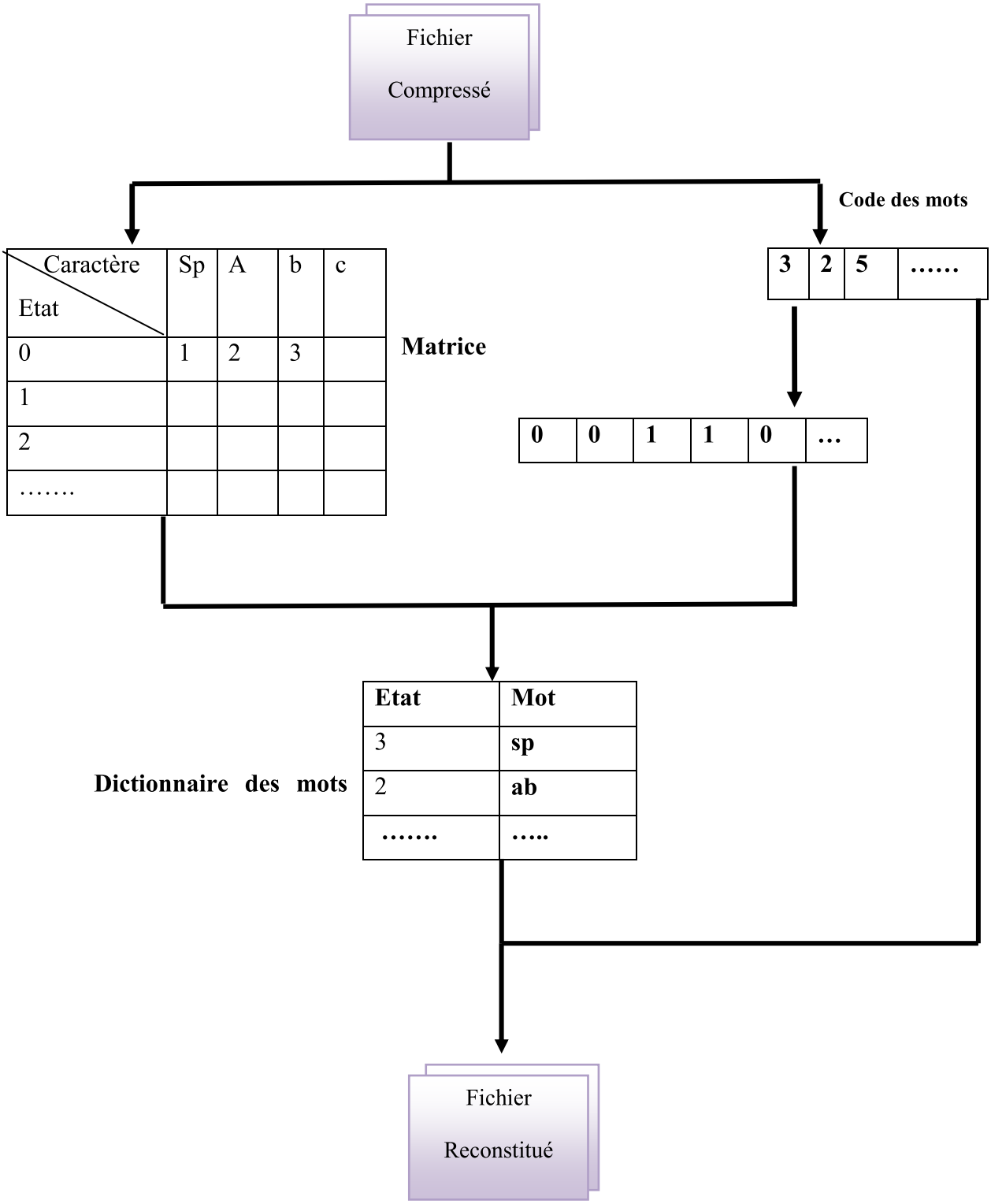


Figure 11. Schéma général de la décompression.

2.1.3. Exemple d'application :

Nous voulons effectuer la compression du message suivant : « bac ca cab baa». Pour cela nous allons suivre les étapes suivantes :

1- Construction de l'automate :

Voici l'automate qui reconnait les mots définit sur l'alphabet suivant : {sp, a, b, c}.

Où : sp représente l'espace.

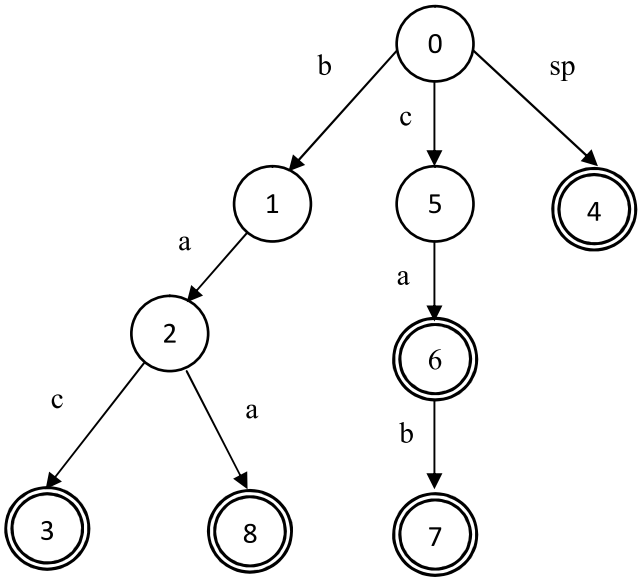


Figure 12. Automate représentant l'exemple d'application.

2- Transformation de l'automate sous forme d'une matrice :

Le tableau suivant correspond à la liste des transitions de l'automate précédent:

- La transition de l'état « 0 » vers l'état « 1 » est régie par un b.
- La transition de l'état « 0 » vers l'état « 4 » est régie par un sp.
- La transition de l'état « 0 » vers l'état « 5 » est régie par un c.
- La transition de l'état « 1 » vers l'état « 2 » est régie par un a.
- …ect

Remarque : s'il n'ya pas de transition on met « - », NBT représente le nombre de transition pour chaque état.

- **La table de transition :**

Etats	NBT	sp	a	b	c
0	3	4	-	1	5
1	1	-	2	-	-
2	2	-	8	-	3
3	0	-	-	-	-
4	0	-	-	-	-
5	1	-	6	-	-
6	1	-	-	7	-
7	0	-	-	-	-
8	0	-	-	-	-

Tableau 4. La table de transition

- **Voici la matrice associée :**

$$\begin{pmatrix} 4 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ 0 & 8 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 13. La matrice associée à l'automate

3- Transformation sous forme d'un arbre linéaire :

Nous allons transformer la matrice précédente à un arbre linéaire en suivant cette démarche qui consiste à parcourir l'automate niveau par niveau :

- **Pour le premier niveau :** l'état initial « 0 » a trois éléments non nuls qui sont « 4 », « 1 » et « 5 ».leurs indices sont respectivement sp, b et c.

0	4	1	5

- L'arbre linéaire obtenue pour le premier niveau est : **3 sp b c**

➢ **Pour le deuxième niveau :** on commence par l'état « 4 » qui est un état final, il n'a pas de transition (0 transition)

0	4	1	5	0

- On passe à l'état « 1 » qui a un seul élément non nul qui est « 2 », son indice est a.

0	4	1	5	0	2

- On passe à l'état « 5 » qui a un seul élément non nul qui est « 6 », son indice est a.

0	4	1	5	0	2	6

- L'arbre linéaire obtenue pour le deuxième niveau est : **0 1 a 1 a**

➢ **Pour le troisième niveau :** on commence par l'état « 2 » qui a deux éléments non nuls qui sont « 8 » et « 3 », leurs indices sont respectivement a et c.

0	4	1	5	0	2	6	8	3

- On passe à l'état « 6 » qui a un seul élément non nul qui est « 7 », son indice est b.

0	4	1	5	0	2	6	8	3	7

- L'arbre linéaire obtenue pour le troisième niveau est : **2 a c 1 b**

- **Pour le dernier niveau :** tout les états sont finaux, alors ne contiennent pas de transitions.

0	4	1	5	0	2	6	~~8~~	~~3~~	~~7~~	0	0	0

- L'arbre linéaire de toute la matrice est : **3 sp b c 0 1 a 1 a 2 a c 1 b 0 0 0**

4- Construction d'une nouvelle matrice :

Nous allons construire une nouvelle matrice associée à l'arbre linéaire.

- **La table de transition :**

Etats	NBT	sp	a	b	c
0	3	1	-	2	3
1	0	-	-	-	-
2	1	-	4	-	-
3	1	-	5	-	-
4	2	-	6	-	7
5	1	-	-	8	-
6	0	-	-	-	-
7	0	-	-	-	-
8	0	-	-	-	-

- **Voici la matrice associée :**

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{2} & \mathbf{3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{4} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{5} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{6} & \mathbf{0} & \mathbf{7} \\ \mathbf{0} & \mathbf{0} & \mathbf{8} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

Figure 14. la matrice associé à l'arbre linéaire

Si on veut construire l'automate à partir de cette nouvelle matrice, on obtient exactement le même que le précédent**.**

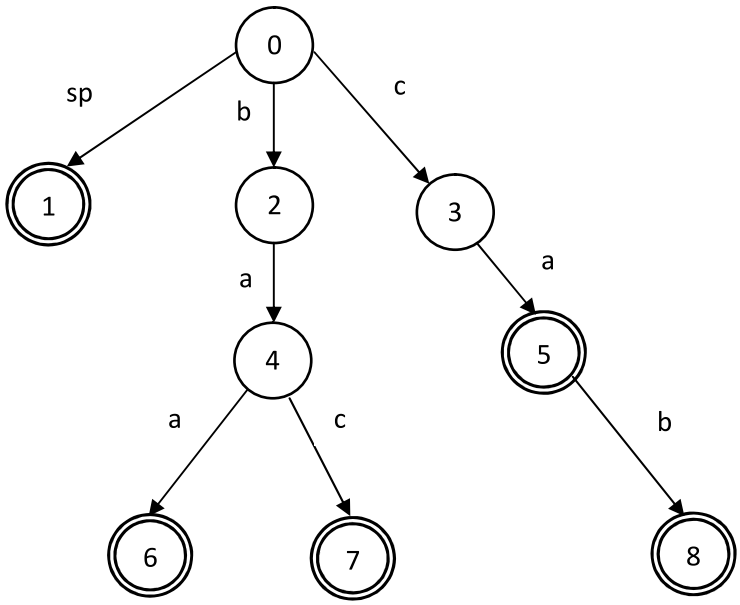


Figure 15. Automate obtenu à partir de la nouvelle matrice

Le résultat de la compression sera les états finaux de tous les mots.

Pour l'exemple « bac ca cab baa» :

- L'état final de **bac** est « 7 »
- L'état final de **sp** est « 1 »
- L'état final de **ca** est « 5 »
- L'état final de **cab** est « 8 »
- L'état final de **baa** est « 6 »

Donc le résultat de la compression est :

7	1	5	1	8	1	6

- **On passe à la décompression :**

1- Récupérer l'arbre linéaire, les codes des mots ainsi que la nouvelle matrice :

- Récupérer l'arbre linaire **: 3 sp b c 0 1 a 1 a 2 a c 1 b 0 0 0.**

- Récupérer les codes des mots :

7	1	5	1	8	1	6

- Reconstruire la nouvelle matrice à partir de l'arbre linaire**:**

Etats	NBT	sp	a	b	c
0	3	1	-	2	3
1	0	-	-	-	-
2	1	-	4	-	-
3	1	-	5	-	-
4	2	-	6	-	7
5	1	-	-	8	-
6	0	-	-	-	-
7	0	-	-	-	-
8	0	-	-	-	-

2- Créer la table « Final »:

Si l'indice *i* de la table **Final** est un état final, alors **Final[i]** reçoit**1**, sinon reçoit **0.**

0	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

Tableau 5. Table « Final»

3- Créer la table mot :

On parcourt la matrice état par état. Dés qu'on trouve un état final, on l'insère dans la table **mot** ainsi que le mot qui lui correspond (le mot constité dans ce parcours).

Etat final	**Mot**
1	Sp
6	Baa
7	Bac
5	Ca
8	cab

Tableau 6. Table « mot »

4- Récupérer le texte source :

On lit les codes des mots un par un.et on récupère le mot qui lui correspond dans la table **mot**.

Voici le résultat de la décompression : **«** bac ca cab baa **».**

3. Conception :

3.1.Classement des approches de conception : [15]

3.1.1. Approche descendante (top-down) :

Par analogie, dans les domaines techniques, cette approche consiste à concevoir le sujet d'études ou le produit dans les grandes lignes, puis, itérativement, à s'intéresser à des détails de plus en plus fins. L'approche *top-down* permet de délimiter et de conceptualiser rapidement le projet et de le diviser en sous-parties aisément manipulables. Elle permet donc d'avoir une vue globale du projet final et de donner une estimation rapide, bien qu'approximative, de sa complexité et de son coût.

3.1.2. Approche ascendante (bottom-up):

Une approche dite ascendante, ou approche *bottom-up* « de bas en haut », se caractérise par une suite de processus qui apportent chacun une partie fondamentale de l'édifice qu'elle cherche à produire, à partir d'éléments de base.

3.2.Modularité :

3.2.1. Notion de module : [21]

Le mot **MODULE** est un des mots les plus employés en programmation moderne, cette notion de module se trouve implantée. B.Meyer est l'un des principaux auteurs avec G.Booch qui ont le plus travaillé sur cette notion. Le premier a implanté ses idées dans le langage orienté objet " Eiffel ", le second a utilisé la modularité du langage Ada pour introduire le concept d'objet qui a été la base de méthodes de conception orientées objet : OOD, HOOD, UML...

3.2.2. Critères principaux de modularité :

Les 5 principes retenus sont :

1. **La décomposition modulaire :**

Capacité de décomposer un problème en sous-problèmes, semblable à la méthode structurée descendante.

2. **La composition modulaire :**

Capacité de recombinaison et de réagencement de modules écrits, semblable à la partie ascendante de la programmation structurée.

3. **La continuité modulaire :**

Capacité à réduire l'impact de changements dans les spécifications à un minimum de modules liés entre eux, et mieux à un seul module.

4. La compréhension modulaire :

Capacité à l'interprétation par un programmeur du fonctionnement d'un module ou d'un ensemble de modules liés, sans avoir à connaître tout le logiciel.

5. La protection modulaire :

Capacité à limiter les effets produits par des incidents lors de l'exécution à un nombre minimal de modules liés entre eux, mieux à un seul module.

3.3. Les différents modules de l'application :

Notre application doit offrir la possibilité de gérer les différentes tâches et demandes de l'utilisateur, à savoir la compression, la décompression, le calcul et l'affichage des résultats et des performances de compression / décompression et éventuellement leurs sauvegarde.

Donc nous pouvons décomposer le noyau de l'application en quatre modules principaux :

1. Le module de gestion de fichiers.
2. Le module de compression.
3. Le module de décompression.
4. Le module des performances.

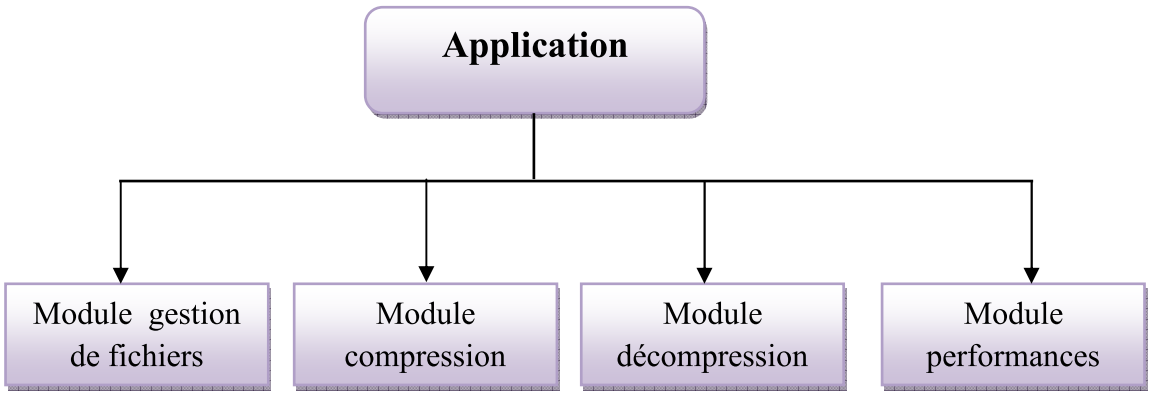


Figure 16. Module global de l'application.

3.3.1. Le module de gestion de fichiers :

Ce module permet de gérer les fichiers manipulés par les autres modules (compression/décompression/calcul des performances).

- Il ouvre le fichier source qui porte l'extension (.txt), c'est le fichier à compresser avec la fonction **Open ( )**.
- Il crée un nouveau fichier avec l'extension (.comp) et l'ouvre en écriture, c'est le fichier qui contiendra le résultat de la compression en utilisant la fonction **Ouvrir ( )**.
- Il crée un autre nouveau fichier avec l'extension (.txt) et l'ouvre en écriture, c'est le fichier qui contiendra le résultat de la décompression.

La figure ci-dessous illustre l'architecture générale du module de gestion de fichiers:

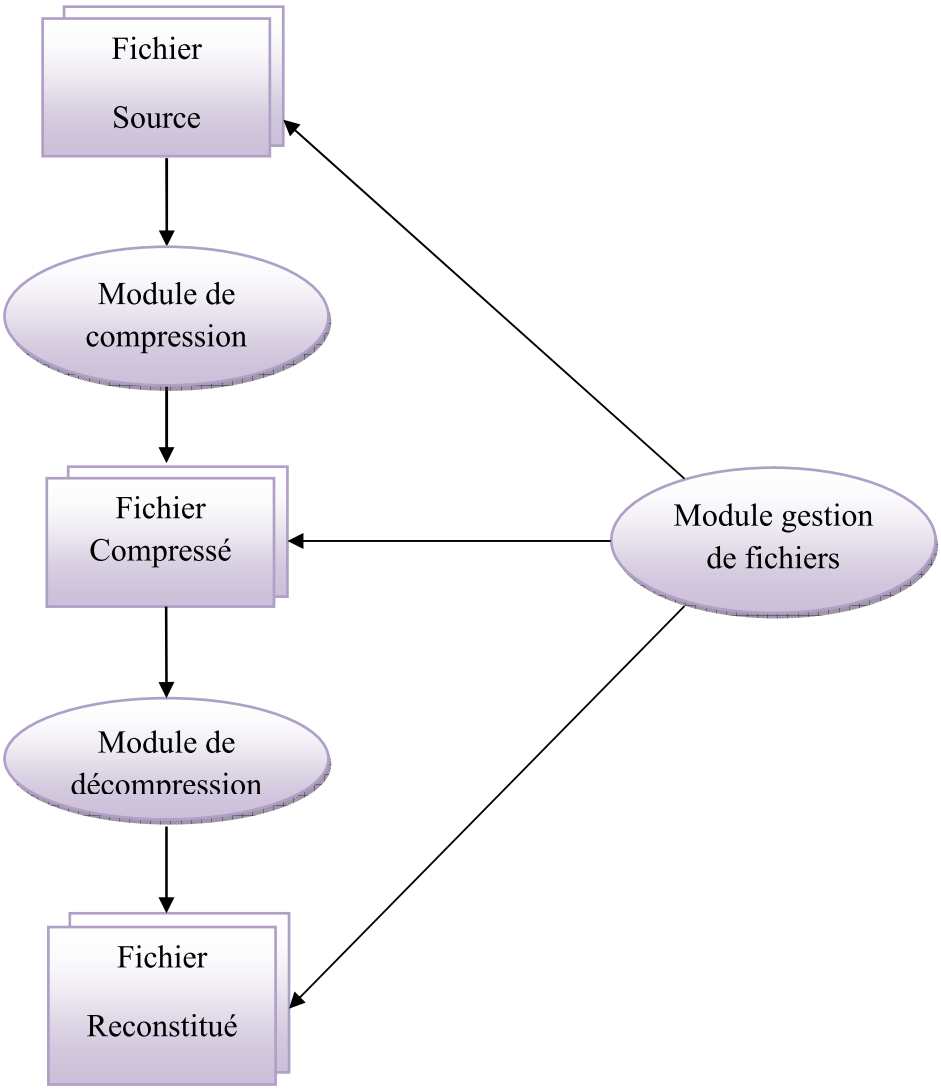


Figure 17. Schéma général du module de gestion des fichiers

3.3.2. Le module de compression :

Ce module se charge de la compression du fichier source.

- La fonction **Construire ( )** permet de construire la matrice initiale, et cela se fait par les étapes suivantes :
  - Elle fait appel à la fonction **Open ( )** qui permet d'ouvrir le fichier source.
  - Ensuite fait appel à la fonction **Get_Mot ( )** qui permet de retourner tous les mots que le fichier contient, c'est l'analyse lexicale.
  - A chaque fois qu'un mot est reconnu, il sera ajouté dans la matrice grâce à la fonction **Ajouter ( )**, et cela en calculant l'état courant et l'état suivant de chaque caractère.
- La fonction **Affect ( )** permet de construire l'arbre linaire à partir de la matrice Initiale.
- La fonction **Arb_To_Mat ( )** permet de construire la nouvelle matrice triée par ordre croissant des états.
- La fonction **compresser ( )** permet de compresser la nouvelle matrice.

La figure ci-dessous illustre l'architecture générale du module de compression :

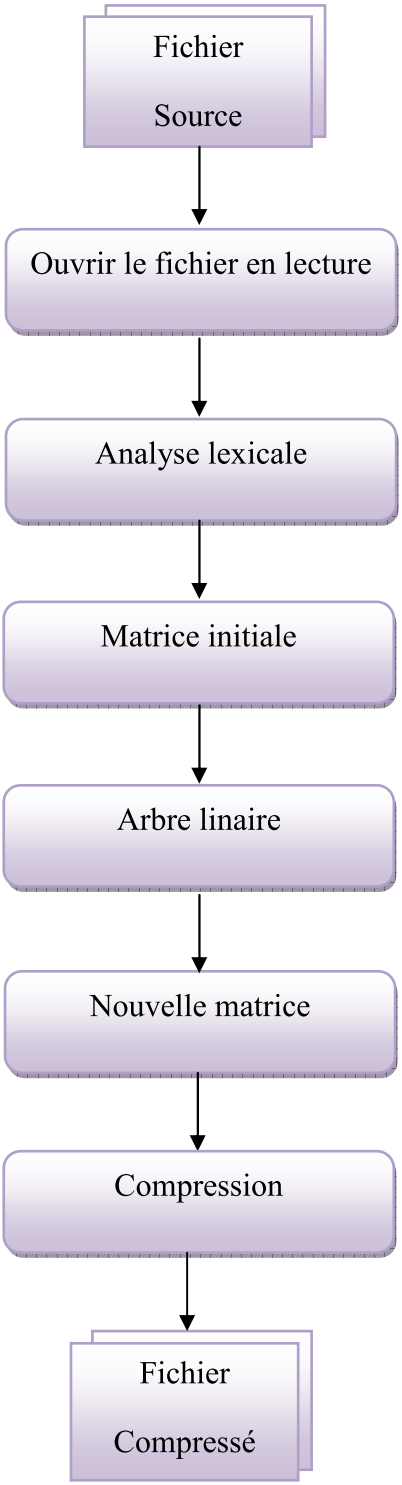


Figure 18. Schéma générale du module de compression

3.3.3. Le module de décompression :

Ce module se charge de la décompression du fichier compressé.

- La fonction **open ( )** ouvre le fichier décompressé en lecture.
- La fonction **ouvrir ( )** ouvre un nouveau fichier en écriture, c'est le fichier qui contiendra le résultat de la décompression.
- La fonction **récupérer ( )** récupère l'arbre linaire et les codes des mots.
- La fonction **Arb_To_Mat ( )** permet de reconstruire la nouvelle matrice à partir de l'arbre linaire.
- La fonction **boolfin ( )** permet de construire la table final.
- La fonction **Get_Dictionnaire ( )** permet de récupérer tous les mots en parcourant la nouvelle matrice.
- La fonction **texte ( )** permet de récupérer le texte et l'écrire dans le fichier.txt.

La figure ci-dessous illustre l'architecture générale du module de décompression :

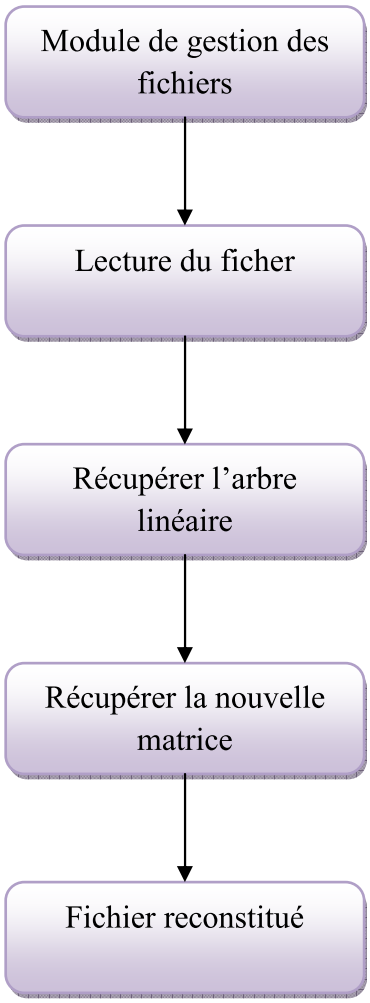


Figure 19. Schéma général du module de décompression.

3.3.4. Module affichage et calcul des performances :

Lors de compression et de décompression y a un module qui charge du calcul des performances de l'algorithme, comme la taille du fichier source et la taille du ficher compressé et le taux de compression.

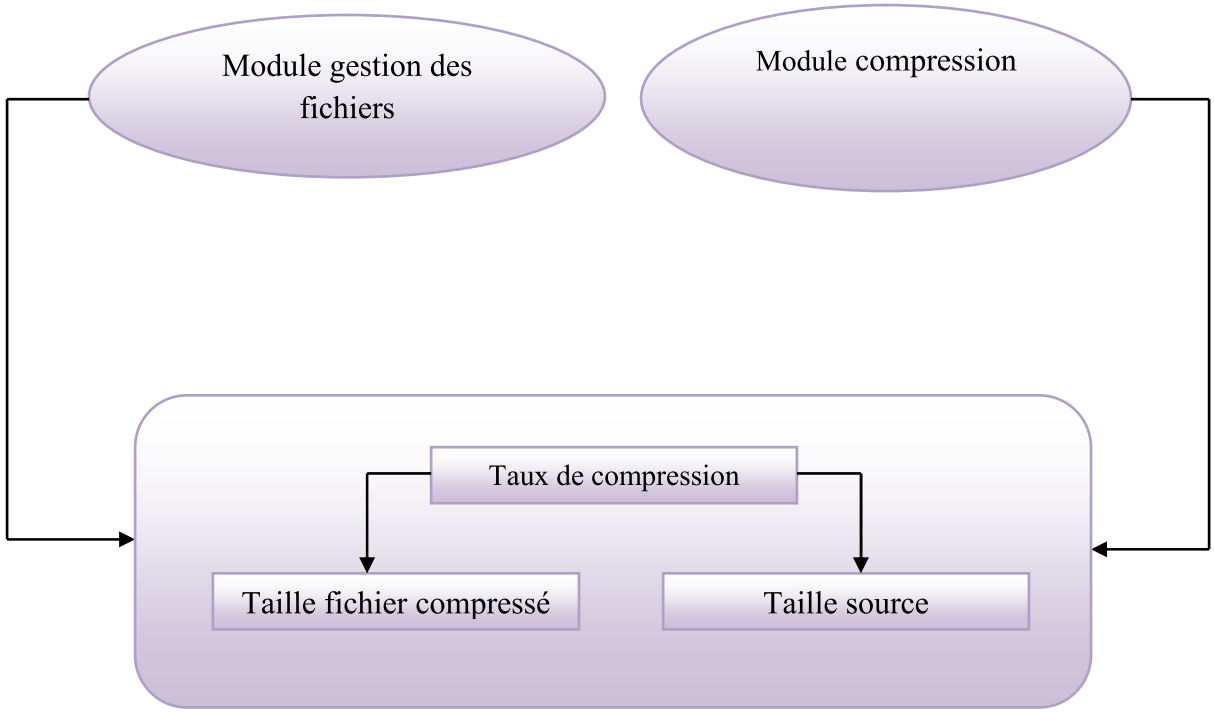


Figure 11. Schéma général du module calcul et affichage des performances

4. Les algorithmes de chaque module :

4.1.Algorithme du module de gestion de fichiers :

Ce module possède deux classes principales qui gèrent les entrées/sorties de l'application.

La fonction **Open ( )** ouvre les fichiers en lecture, et la fonction **Ouvrir ( )** permet d'ouvrir les fichiers en écriture.

Début

Si (compression)

- Entrer le nom du fichier qui porte l'extension (.txt)
- Vérifier :
  - **Si** le fichier existe alors :
    - Ouverture du fichier source en lecture.
    - Calculer la taille de ce dernier.
    - Lecture du fichier mot par mot via l'analyseur lexical.
    - Fermeture du fichier source.

    Finsi

- Créer le nouveau fichier qui porte l'extension (.comp)
- Vérifier :
  - **Si** le fichier est crée alors :
    - Ouverture du fichier en écriture.
    - Sauvegarder le résultat de la compression dans le fichier.
    - Fermeture du fichier

    Finsi.

Finsi

Fin

Début

Si (Décompression)

- Entrer le nom du fichier compressé et son extension
- Vérifier :
  - **Si** (fichier existe) alors
    - ouverture du fichier en lecture.
    - Lecture du fichier compressé au fur et à mesure pour le décompresser.
    - Fermeture du fichier compressé.

    Finsi

- Créer un nouveau fichier qui porte l'extension (.txt)
- Vérifier :
  - **Si** le fichier est crée alors :
    - Ouverture du fichier en écriture.
    - Sauvegarder le résultat de la décompression dans le fichier.
    - Fermeture du fichier

    Finsi

Finsi

Fin

4.2.Algorithme du module de compression :

Début

- Ouvrir le fichier source en lecture.
- Créer un nouveau fichier qui contiendra le résulta de la compression.

//construire la matrice initiale

Tant qu'on n'est pas à la fin du fichier **faire**

- Lire le fichier source mot par mot via l'analyseur lexical.
- Construire la matrice initiale à deux dimensions (état/caractère) :
  - **Tant que** ce n'est pas la fin du mot **faire**
    - Lire le mot caractère par caractère.
    - Etat suivant reçoit matrice [état courant][caractère].
    - **Si** (état suivant égale à zéro) alors
      - ✓ Etat suivant reçoit le nombre d'état.
      - ✓ matrice [état courant][caractère] reçoit état suivant.
      - ✓ Incrémenter le nombre de fils de l'état courant.
      - ✓ Incrémenter le nombre d'état.

      Finsi
      - ✓ Incrémenter l'état courant.

    Fin tant que

Fin tant que

- Fermeture du fichier source.

// Construire l'arbre linéaire

Si la position de l'état courant égale la position de l'état fin **alors**

Retourne 0.

Sinon

- Récupérer le tableau de nombre de fils pour chaque état nœud.
- **Tant que** le nombre de caractère inférieur à 256 **Faire**
  - Valeur reçoit matrice [état courant][caractère].
  - **Si** la valeur n'est pas nulle **alors**
    - Arbre plat reçoit caractère.
    - Tableau d'états finaux reçoit valeur.
    - Incrémenter la position de l'état fin.

    Finsi

Fin tant que

- Incrémenter la position de l'état courant.
- Refaire la même chose pour chaque état courant.

Finsi.

//Construire la nouvelle matrice

- Initialiser la nouvelle matrice à zéro.
- Initialiser la position de l'arbre plat à zéro.
- Initialiser le compteur à 1.
- **Pour** i allons de 0 **jusqu'à** nombre d'état **faire**
  - **Pour** j allons de 0 **jusqu'à** nombre de fils de l'état i **faire**
    - Nouvelle Matrice[i][arbre plat [position]] reçoit compteur.
    - Incrémenter le compteur.
    - Incrémenter la position de l'arbre.

  Fin pour

Fin pour

//compression de la matrice

Tant que ce n'est pas la fin du mot **faire**

- Lire le mot caractère par caractère.
- Etat reçoit nouvelle Matrice [état][car].

Fin tant que

- Retourner état final du mot lu.

Fin

4.3.Algorithme de module de décompression :

Début

- Ouvrir le fichier à décompresser en lecture.
- Ouvrir un nouveau fichier texte qui contiendra le résultat de la décompression.
- Récupérer les données enregistrées dans l'entête du fichier compressé :

Tant que le fichier n'est pas vide **faire** :

- Lire le nombre d'états.
- Lire le nombre de mots.
- Lire le tableau de caractères.
- Lire le tableau de nombre de fils pour chaque état.
- Lire le code fin de chaque mot.

Fin tant que

- Reconstruire la nouvelle matrice à partir des données lues.
- **Pour** i allons de 0 jusqu'à nombre d'états **faire**
  - Initialiser le tableau des états finaux.
- **Pour** i allons de 0 **jusqu'au** nombre de mots **faire**
  - **Pour** j allons de 0 **jusqu'au** nombre d'états **faire**
    - Si (code du mot égal à j)
    - Etat final de j reçoit 1.

  Fin pour

Fin pour

- Afficher le tableau de booléen.
- Initialiser le dictionnaire les mots.
- **Pour** i allons de 0 **jusqu'au** nombre de mot **faire**
  - Lire le tableau des codes.
  - Récupérer le mot correspondant au code.
  - Ecrire le mot dans le dictionnaire des mots.
  - Ecrire la liste des mots dans le fichier ouvert en écriture.

Fin pour

- Fermer le nouveau fichier.

Fin

4.4.algorithme du module affichage et calcul des performances :

- **Calculer la taille du fichier :**

Début

- Ouvrir le fichier en lecture.
- Positionner le curseur de début jusqu'à la fin du fichier.
- Récupérer la taille du fichier qui est la position du curseur.

Fin

- **Calcul du taux de compression**

Début

- Récupérer la taille du fichier original.
- Récupérer la taille du fichier compressé.
- Taux = (Taille_Fich_Comp/Taille_Fich_Orig)*100.

Fin

Conclusion :

Dans ce chapitre nous avons d'abord présenté la méthode de compression basée sur les automates d'états finis. Puis, nous avons réalisé une analyse et une conception d'un système de compression/décompression, ensuite nous avons présenté les différents modules qui constituent l'application. Enfin, dans la conception nous avons spécifié d'une façon formelle les différents algorithmes et organigrammes qui composent ce système de compression/ décompression.

Chapitre IV

Implémentation & évaluation

1. Introduction :

Dans ce chapitre nous allons passer à la réalisation de notre application en se basant sur les notions et les concepts présentés dans la phase de la conception.

Pour ce faire, notre choix est porté sur un environnement graphique et visuel qui est le Windows, un langage de programmation qui est le C++, et un environnement de développement orienté objet qui est le Dev C++.

2. l'environnement technique de développement :

La réalisation de notre application a été faite sur un micro-ordinateur ayant les caractéristiques suivantes :

Processeur : Intel(R),Core(TM)2 Duo CPU T6670@ 2.20 GHz 2,20 GHz

Mémoire vive : 4,00 GO

Type du système : Système d'exploitation 32 bits

Pour les logiciels utilisés, nous avons opté pour :

- Windows pour le système d'exploitation.
- Dev C++ comme environnement de développement.

2.1. Présentation de DEV C++ : [15]

Le logiciel **Dev-C++** est un environnement de développement intégré permettant de programmer en C/C++. Il utilise la version MinGW du compilateur GCC (*venu du monde du logiciel libre*) et permet d'exporter ses projets sous fichiers .dev.

Le compilateur Dev-C++ est assez complet. Il comprend entre autre un "répertoire de classes", un "répertoire de fonctions incluses", et un débogueur qui permet de surveiller l'état des variables pendant l'exécution du programme. L'avantage de **Dev-c++**, c'est qu'il est multiplateforme. Il permet de générer des exécutables fonctionnels sous la majorité des plateformes. L'outil de développement Dev-c++ peut être très pratique. Il est important de savoir s'en servir efficacement.

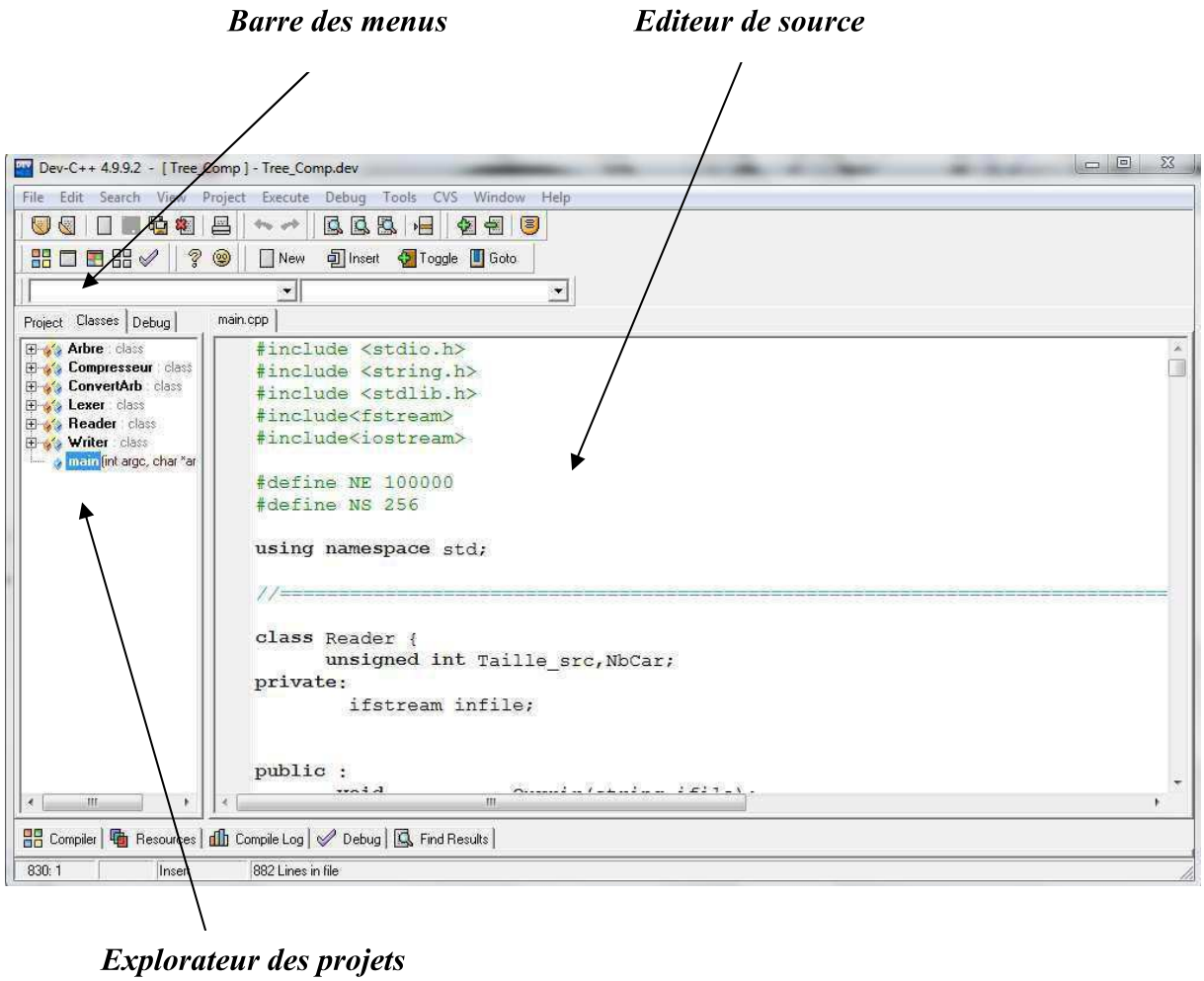


Figure21 .Interface de l'environnement Dev C++

2.2.Présentation du langage de programmation C++ :

Le C++ est l'un des langages de programmation les plus utilisés actuellement vu son efficacité.

Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, ce langage est, avec le C, idéal pour ceux qui doivent assurer la portabilité de leurs programmes au niveau des fichiers sources (pas des exécutables).

Le C++ est un langage orienté objet car il répond aux trois principes fondamentaux : encapsulation, polymorphisme et héritage. En plus, des types simples et complexes du C, on peut définir des classes qui sont des structures évoluées dans certains champs (appelées membres) sont des fonctions (appelées méthodes) selon le principe d'encapsulation. Le programmeur doit gérer l'accessibilité aux champs des classes qu'il définit : certains membres peuvent être rendus accessibles directement comme un champ de structure.

Les principaux avantages du C++ sont les suivants :

- Grand nombre de fonctionnalités.
- Performances du C.
- Facilité d'utilisation des langages objets.
- Portabilité des fichiers sources.
- Facilité de conversion des programmes C en C++, et, en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C.
- Contrôle d'erreurs accrues.

3. Présentation de l'application :

Dans cette partie, on va expliquer le fonctionnement de notre application, et quelle sont les commandes pour son exécution**.**

3.1. Le choix du fichier à compresser/décompresser :

Pour compresser ou décompresser un fichier il faut que ce fichier et l'exécutable de l'application se trouvent dans le même répertoire. L'exécution de l'application se passe sur l'invite de commande.

3.2.Compression :

3.2.1. Syntaxe de la commande de compression :

Pour lancer la compression en tape la commande suivante :

…/>compression.exe Fichier_texte Fichier_compressé

Exemple :

C:\Users\PB\Desktop\Compression>compression.exe bible.txt bible.comp

Où:

- **C:\Users\PB\Desktop\Compression>:**le chemin de l'exécutable ainsi que le fichier à compresser.
- **compression.exe :** le nom de l'exécutable.
- **bible.txt:** le nom du fichier à compresser.
- **bible.comp:** le nom du fichier compressé.

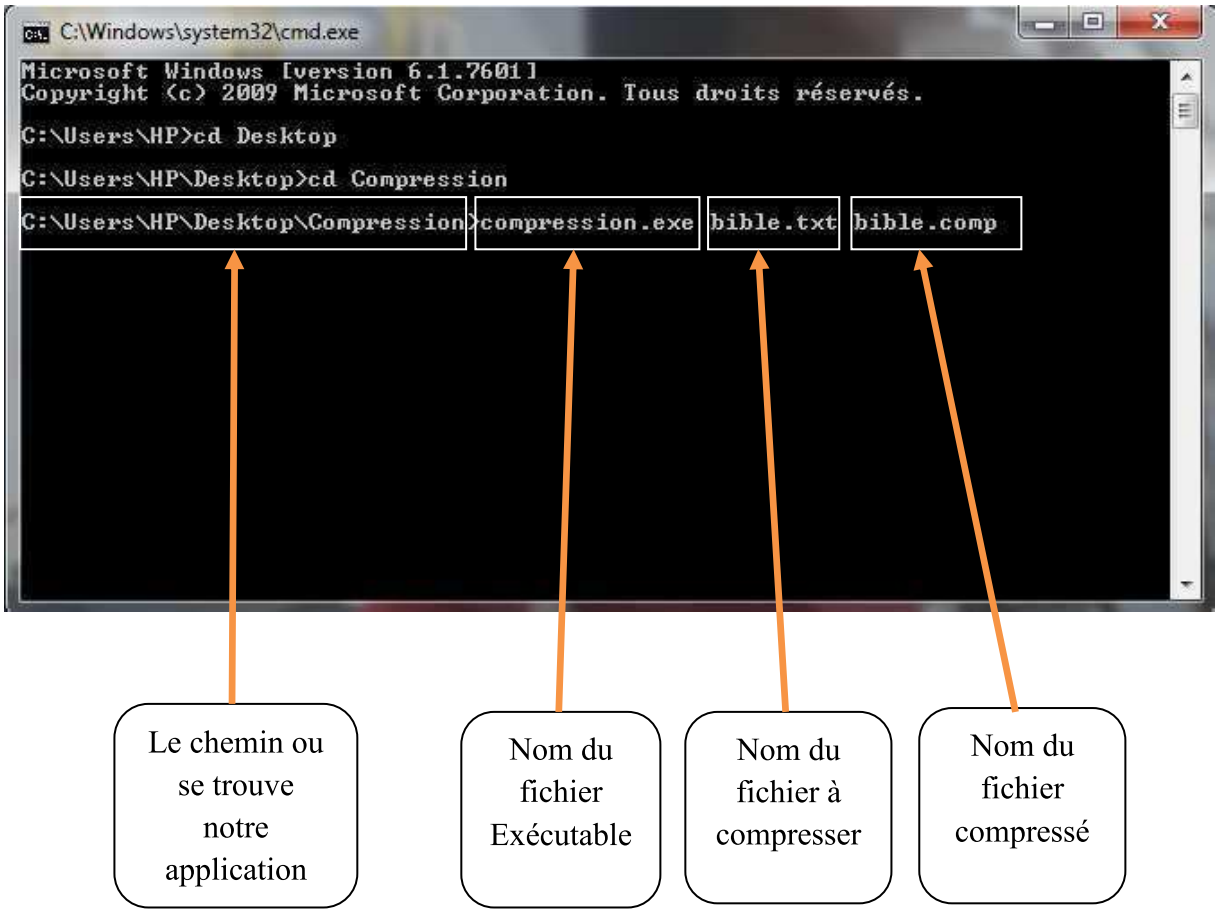


Figure 22.Lancement de la compression

3.2.2. Les résultats de la compression :

Après le lancement de la compression nous aurons les résultats suivants :

- ✓ La taille du fichier source.
- ✓ La taille du fichier compressé.
- ✓ Le taux de compression.
- ✓ Le temps de compression.

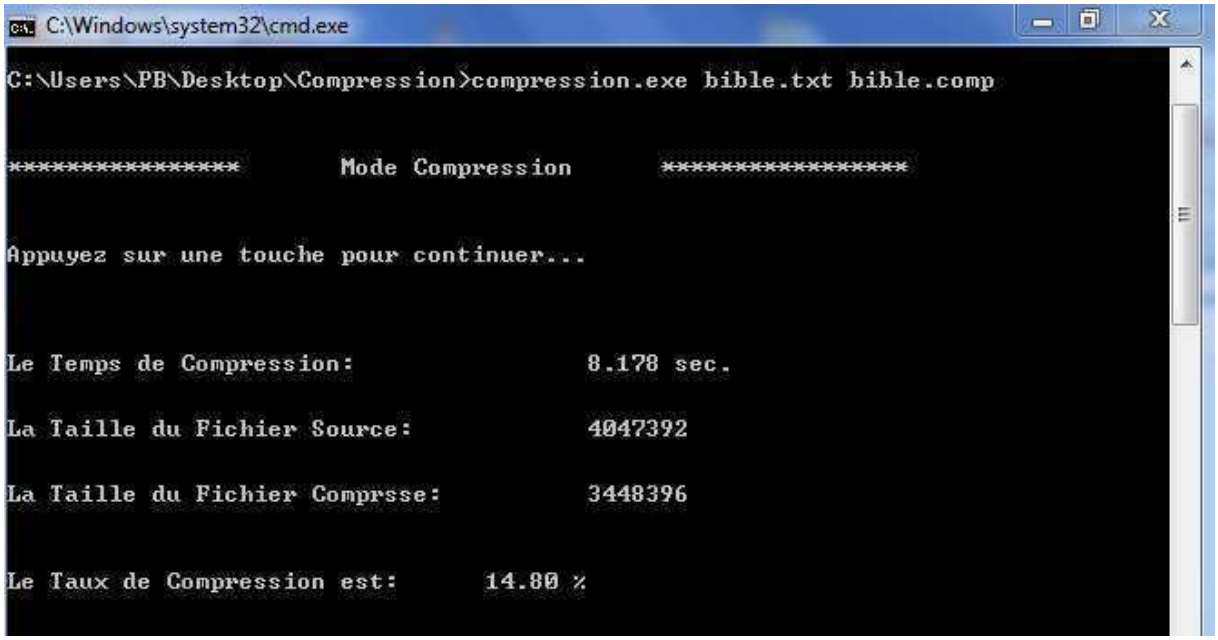


Figure 23. Les résultats de la compression.

3.3. Décompression :

3.3.1. Syntaxe de la commande de décompression :

Pour lancer la décompression en tape la commande suivante :

…/>decompression.exe Fichier_compressé Fichier_texte

Exemple :

C:\Users\PB\Desktop\Deompression>decompression.exe bible.comp bible.txt

Où:

- **C:\Users\PB\Desktop\Decompression>:**le chemin de l'exécutable ainsi que le fichier à décompresser.
- **decompression.exe :** le nom de l'exécutable.
- **bible.comp:** le nom du fichier à décompresser.
- **bible.txt:** le nom du fichier reconstitué.

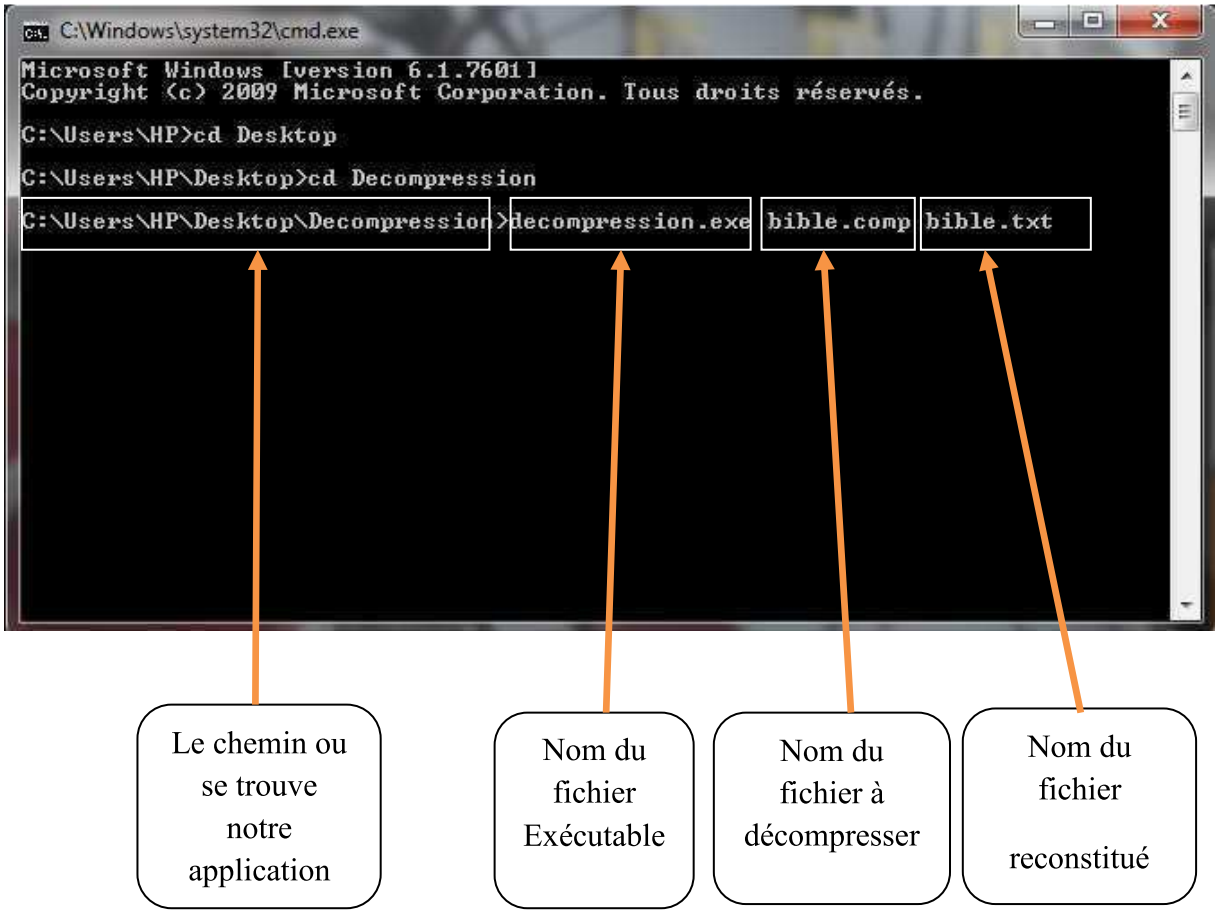


Figure 24.Lancement de la décompression

3.3.2. le résultat de la décompression :

Après le lancement de la décompression nous aurons le résultat suivant :

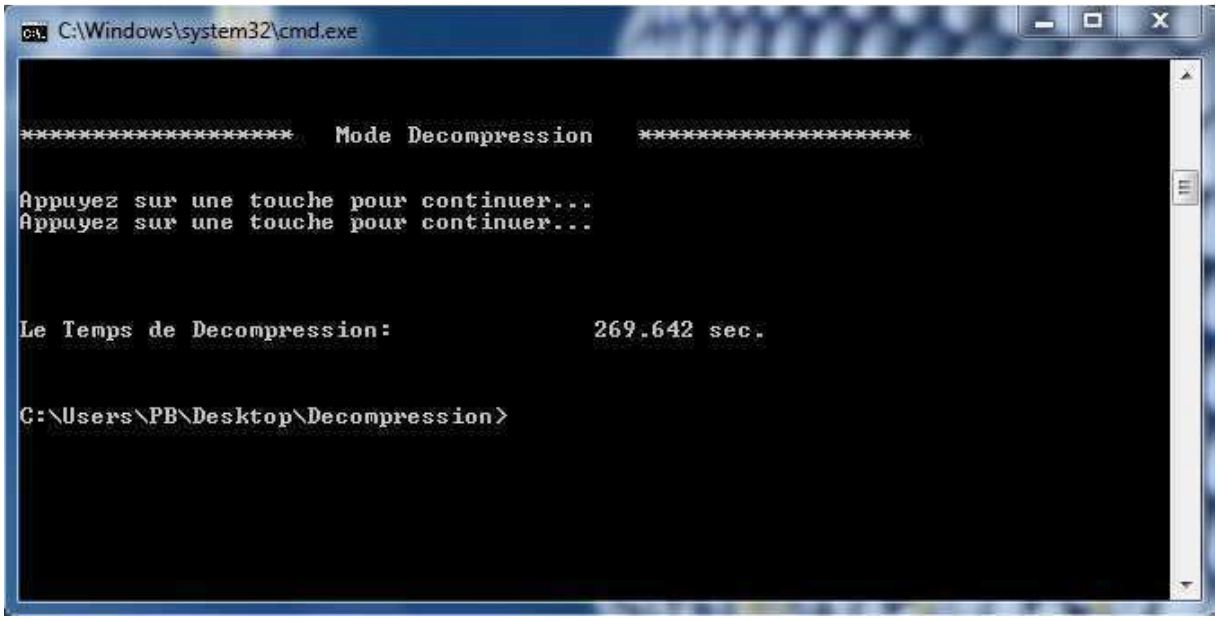


Figure 25. Le résultat de la décompression.

4. Evaluation de l'application :

- **Les fichiers textes :**

Pour évaluer le taux de compression, nous allons faire des testes sur des fichiers textes, ce taux est calculé par la fonction suivante :

$$\textbf{Taux de compression} = 1 - \left(\frac{\text{Taille\_Fichie\_Compressé}}{\text{Taille\_Fichier\_Source}} \right) * 100$$

Nom du fichier	Taille du fichier source (KO)	Taille du fichier compressé (KO)	Taux de compression(%)
Dickens	9954	9264	6.93
Xml	5220	3514	32.69
Bible.txt	3953	3368	14.80
world192.txt	2416	2328	3.64
Book1	751	695	7.55
Book2	597	499	16.47
Plrabn12.txt	471	432	8.39
Lcet10.txt	417	344	17.69
alice29	149	140	6.18
Asyoulik.txt	123	114	7.16
Paper2	81	71	12.25
Paper1	52	48	8.99

Tableau 7. Evaluation de taux de compression sur des fichiers textes.

- On remarque que le taux de compression est entre **3%** et **33%.**
- Le taux de compression moyen est de **11.90%.**

5. Conclusion

Cette partie est consacrée pour la présentation de l'environnement de développement, ainsi le langage de programmation utilisé, par la suite nous avons fait des testes sur un ensemble de fichiers fréquemment utilisés dans la compression des données pour comparer les techniques de compression entre elles. Le corpus sur lequel nous avons évalué notre application est composé de douze fichiers textes de tailles différentes pour permettre de comparer le taux de compression selon la variante de la taille du fichier.

Conclusion générale

Conclusion générale

La compression de données a encore de beaux jours devant elle. Elle permet de résoudre des problèmes de volume, de stockage et de débit. De nombreux travaux de recherche sont en cours afin d'améliorer les performances des techniques de compression.

Notre travail avait comme objectif d'implémenter et d'évaluer une nouvelle méthode de compression des fichiers textes, basée sur des automates à états finis afin de réduire leurs volumes pour gagner d'espace mémoire.

A la fin de notre travail, nous avons constaté que ce nouvel algorithme de compression de données donne de bons résultats de taux et de temps de compression pour les fichiers textes. Malheureusement nous n'avons pas évalué le taux de compression de cette méthode par rapport à une autre déjà existante.

Cette expérience nous a permis d'acquérir des connaissances dans un domaine aussi vaste qui est la compression de données, les différents algorithmes utilisés, ainsi de mieux se familiariser avec le langage de programmation le C++ et de l'outil de développement

DEV C++.

Nous précisons que les différentes parties de notre application peuvent être maintenus et optimiser pour donner encore de bons et de meilleurs résultats.

Références bibliographiques

Références bibliographiques

[1] Codage, cryptographie et application ; Bruno Martin ; Presse polytechnique et universitaire romandes, 2004.

[2] http://www.iich.ch/Tcom/cours/PDF/Compression.pdf.

[3] MM. S. Maadi, Y.Peneveyre, et C. Lambercy « Compression de données avec perte ».

[4] THEORIE DESCODES : compression, cryptage, correction(2005) par Yves Denneulin, Jean- Guillaune Durmas…

[5] Théorie de l'information codage source-canal ESCPI-CNAM version 24 Janvier 2007.

[6] « Compression de données », PEREIRA Vincent - LEPRETTE Franck - HACAULT Vincent, Décembre 2004.

[7] http://perso.iut-nimes.fr/flouchet/DSP/TP04-05.PDF.

[8] L.Mokrane et L.Ramdani : « Compression de données à l'aide des automates d'états finis déterministes», mémoire de Master, UMMTO 2013.

[9] Alexandre THIL-Master1-Informatique-univercité de Metz, Initition à la recherche Algorithmes de compression de données et de traitement d'images.

[10] Mark Nelson. Compression de données (image, son, texte). Edition Dunode 1992.

[11] http://www.dil.univ-mrs.fr/~gcolas/algo-licence/slides/huffman-slides.pdf.

[12] http://webia.lip6.fr/~lepape/ens/isn/dokuwiki/lib/exe/fetch.php?media=wiki:seance6:cauty_codagehuffman_.pdf

[13] http://www.iict.ch/Tcom/Cours/PDF/Compression.pdf.

[14] Polastro Hadrien, *Principes et techniques de compression de données à la portée de tous,* 2007.

[15] Wikipédia.

[16] S.Aliche et S.Saidj : « Implémentation et évaluation de l'algorithme de Huffman pour la compression de données», mémoire de licence, UMMTO 2011.

[17] Alexandre Pauchet, *Modélisation des Systèmes Complexes,Introduction aux Automates à états finis,* 2011.

[18] Tounsi L, Lenté C. and Maurel D. (2006) ; Analyse statistique de la structure des automates représentant des dictionnaires électroniques. In *prod. Of* JADT'06, pages 928, 928.

[19] Tounsi L.(2007). Sous-automates à nombre fini d'etats. Application à la compression de dictionnaire électrique. Thése de Doctorat, Université François Rebelais de Tours, page 46.

[20] http://www.algopt.fr/index.php?option=com_content&view=article&id=79:algorithme-lzw&catid=36:algorithmes&Itemid=57.

[21] http://rmdiscala.developpez.com/cours/LesChapitres.html/Cours4/Chap4.3.htm