PROJET HUFFMAN

JANNÈS Lucas, CHOUKROUN Simon, GUIRRE Bilal, SYLLA Hassan, BERDOUZ Nassim

L2, Promo 2024, Groupe D

Noreddine EUTAMENE

Table des matières

I.Présentation projet

1.Le principe de la compression de fichiers :

La compression de fichiers consiste à modifier son format, afin que celui-ci prenne moins de place en mémoire. Cela permet de stocker plus de choses, dans un espace mémoire plus petit. La compression de fichier présente un autre avantage, en effet un fichier compressé est moins volumineux et donc il peut être envoyé plus rapidement. Aujourd'hui, nous avons tous recours à la compression de fichier sans même nécessairement le savoir. Par exemple lorsque nous passons de PDF à Word ou inversement, une compression à lieu. En bref, la compression permet de rendre un fichier moins volumineux. Dans ce projet, nous allons nous intéresser à la compression de fichier texte grâce au codage Huffman.

2. Le projet Huffman , que permet-il ?

Le projet Huffman nous permet de nous intéresser sur l'arbre Huffman afin de compresser des fichiers textes. Le codage de Huffman est une méthode de compression de données statistiques qui peut réduire la longueur de codage de caractère. Le code de Huffman est le meilleur code de longueur variable, c'est-à-dire que la longueur moyenne du texte encodé est la plus petite. Le codage Hufffman intervient sur le nombre de caractère étant codé sur 8 bits. Lorsque nous voulons le compresser c'est pour le rendre le plus petit possible. L'objectif est de réduire au maximum la taille des caractères, nous allons donc chercher à les réunir entre eux. Pour ce faire, nous allons compter le nombre d'occurrences de chaque caractère. Nous allons ensuite stocker chaque caractère avec son occurrence dans une liste. Il faut donc réussir à les réunir entre eux le plus possible. Nous allons également créer un dictionnaire, qui contient le moyen de stockage de chaque caractère sur 8 bits (un octet), en fonction de son nombre d'occurrence.

3. Organisation

Pour réaliser le projet, nous avons mis en place différentes étapes de codage données par le sujet du projet Huffman. Afin d'être plus efficace, nous avons mis en application les méthodes vues en Génie Logiciel, ce qui nous permet de ne pas perdre de temps entre chaque séquence de travail, tant par rapport à l'organisation du travail, que sur la programmation. Nous avons donc créé plusieurs fichiers, « .h » et « .c », afin de bien séparer nos différentes fonctions. Nous avons aussi été obligé de

créer plusieurs structures de données. Nous avons donc procédé méthodiquement afin de ne pas être perdu dans nos propres fichiers.

II. Structure et fonctions utiles

1.Les structures :

Dans notre projet, nous avons créé différentes structures, qui nous ont permis de stocker facilement des informations, faciliter l'écriture de certaines fonctions et gérer nos informations. Nous avons plusieurs structures, nommées logiquement en fonction de leur rôle afin de s'y retrouver facilement. Cela permet de ne pas se perdre dans nos structures.

a.Struct Element :

L'objectif est de compresser un fichier texte. Nous avons commencé par créer une fonction permettant de parcourir un fichier.txt, ici nommé par exemple Alice.txt. Ensuite, une fonction va parcourir ce fichier, et va stocker dans la structure le code ASCII des caractères ainsi que leur nombre d'occurrences. La fonction va donc parcourir chaque caractère un par un.

```
typedef struct Element{
    int lettre;
    int occu;
    struct Element* next;
}Element;
```

La structure est donc composée de plusieurs éléments. Tout d'abord nous avons une variable lettre de type entier. Ce premier élément va stocker la lettre sous forme de code ASCII, c'est pourquoi c'est un entier et non un caractère.

En dessous, nous avons une variable occu de type entier. Une fois que nous avons parcouru tout le fichier (jusqu'à EOF), la fonction aura compter le nombre de fois que la lettre apparait dans le texte, et va la stocker dans cette variable d'entier occu.

Finalement, le next est un pointeur de type structure. Cela signifie que son type est « Element ». Elle pointe vers l'élément suivant de la liste qui lui aussi est de type « Element ». C'est donc une liste simplement chainée qui permet de faire la liaison entre tous les éléments. Nous pouvons donc trouver ici toutes les lettres du fichier parcouru, ainsi que le nombre de répétition de ses lettres.

b.Struct Nœud :

Cette structure nous permettra de créer l'arbre d'Huffman. En effet, une fois que toutes les lettres et caractères d'un fichier sont stockés dans une liste chainée comme nous l'avons vu précédemment,

nous pouvons créer un arbre, en fonction de leur nombre d'occurrence. C'est un arbre d'Huffman. Nous allons donc simplement transformer la struct element en struct nœud.

```
typedef struct Noeud{
    int lettre;
    int occu;
    struct Noeud* left;
    struct Noeud* right;
}Noeud;
```

Nous nous retrouvons donc avec une structure presque similaire a la précédente, a quelques exceptions. Nous avons bien int lettre et int occu, mais nous avons left et right au lieu d'un next. Au lieu d'avoir une liste chainée, nous allons vouloir créer un arbre binaire, c'est pourquoi nous avons besoin d'un left et d'un right, qui sont, de la même manière qu'un next, des pointeurs de type Nœud. Right et left peuvent pointer vers NULL, ou vers une autre structure du même type. Cette structure va avoir une grande importance dans la création de l'abre de Huffman.

c.Struct ElementN :

Cette structure a un rôle plus subtile a comprendre. Nous avons fais ici une structure de structure. C'est-à-dire que avant de vouloir créer l'arbre de huffman, nous devons passer par une étape intermediaire. En effet l'objectif est de pouvoir ordonner la structure précedente sous forme de liste. Une fonction viendra ensuite les assembler pour créer l'arbre de Huffman.

```
typedef struct ElementN{
    Noeud* node;
    struct ElementN* next;
}ElementN;
```

Ici, nous retrouvons donc node qui est de type nœud (la structure precedente). Cette variable va donc simplement contenir une structure de type nœud, comprenant donc toutes ses variables. Ensuite nous avons un next de type ElementN*. « * » signifie que c'est un pointeur sur une structure de type ElementN. Son rôle va être de relier differentes structures entre elles, de manière à avoir une liste chainée.

d.Struct PileElement :

Cette structure intervient après la création de l'arbre de huffman. Son rôle est primordial. Comme son nom l'indique c'est une Pile, qui va servir à stocker les chemins dans l'arbre de huffman. On rappelle que dans notre arbre, le parcours du chemin de gauche est égal a 0, et le chemin de droit est égal à 1. Lorsque nous cherchons un caractère nous parcourons un chemin, par exemple « 0100 », et nous stackons ce nombre binaire dans la Pile. Une fois que nous trouvons un caractère dans l'abre, nous l'écrivons dans un fichier txt, préalablement ouvert grâce à la fonction « fopen » et sa méthode « w » (write), pour écrire dedans. Nous écrivons donc le caractère, suivi de son nombre binaire correspondant. On sait également que plus la suite binaire est courte, plus le caractère apparait souvent dans notre fichier (c'est tout le principe de la compression). Pourquoi une Pile ? Car cela nous permet de pouvoir dépiler le dernier chiffre, pour pouvoir tester d'autres chemins aux alentours. Cela nous permet donc d'être sûr de ne pas oublier de caractère.

```
typedef struct PileElement{
    char car;
    struct PileElement* next;
}PileElement;
```

Nous avons donc un char car, qui est donc un caractère. Dans cette variable sera stocker soit 1, soit 0, en fonction du chemin que l'on va empiler et dépiler. Struct PileElement* est donc un pointeur sur une structure de même type. Cela va donc permettre de pouvoir aligner des chemins plus ou moins long (011010, ou 011) par exemple.

e.Struct ElementD :

Cette structure sert à la fin du programme. Elle va permettre de stocker une lettre, ainsi que son occurrence sous forme de chaine de caractère par exemple : « 010010 ». Durant l'encodage, nous allons effectuer le début du programme dans le sens inverse. Nous allons donc avoir le caractère ainsi que son occurrence en binaire.

```
typedef struct ElementD{
    int lettre;
    char* occu;
    struct ElementD* next;
}ElementD;
```

Int lettre stocke la lettre en binaire, avec char* occu, son occurrence et pour finir nous avons struct ElementD* qui est un pointeur vers une structure de même type afin d'avoir une liste chainée.

2. Les fonctions principales

Tout d'abord, la première fonction à réaliser doit pouvoir traduire un fichier texte, de caractère en un fichier binaire, avec des chiffres (0 et 1). Nous avons donc fait le choix de retourner un char*, (bin), qui représente le caractère sous forme de son nombre ASCII. Nous avons donc créé un tableau d'entier de 8 valeurs, pour stocker 1 octet de 8 bits. Le but de cette fonction est donc de retourner un char en son code ASCII.

Nous avons ensuite créé une fonction secondaire, « lecture_ecriture », qui va appeler la fonction précédente. Pour ce faire, nous ouvrons un fichier, avec un texte en mode « r » pour read, et un fichier vierge en mode w, pour pouvoir écrire dedans et traduire le fichier texte en un fichier binaire. Cette fonction va donc appeler la précédente, et parcourir tout le fichier texte en s'arrêtant sur chaque caractère pour le traduire en binaire, et ainsi l'écrire sur le fichier vierge. Nous nous arrêtons dès lors que nous trouvons EOF, qui signifie end of file, pour fin du fichier en français.

Une fois la fonction effectuée, nous pouvons voir dans notre dossier, un nouveau fichier comprenant tout le fichier traduit en binaire avec le code ASCII de chaque caractère.

Nous avons une autre fonction, permettant de compte le nombre de caractère d'un fichier texte. De la même manière que les fonctions précédentes, nous parcourons tous le fichier texte et nous nous

arrêtons à End Of File. A chaque fois que nous parcourons un caractère, nous avons créé et initialiser un int à 0, et nous l'incrémentons a chaque fois que nous trouvons un nouveau caractère.

Ensuite, nous avons une fonction qui renvoie une liste contenant son chaque caractère ainsi que son nombre d'occurrences. Pour ce faire, nous avons fais le choix de faire 2 fonctions pour atteindre cet objectif.

Tout d'abord nous avons insert_list, qui retourne un pointeur vers une structure Element. Cette fonction va remplir UNE structure de la sorte : nous stockons un caractère (le premier ou un autre), dans la case de la structure dédié a cet usage. Nous allons ensuite dire occurrence = 1 sans même parcourir le fichier pour compter son nombre d'occurrence, nous allons voire cela dans la 2eme fonction

Nous allons ensuite parcourir le fichier, de la même manière que les fonctions précédentes, de manière à compter le nombre d'occurrences du caractère sélectionné. A chaque fois que nous trouvons ce caractère, nous incrémentons l'occurrence. Cette fonction présente également une autre fonctionnalité. En effet cette fonction est capable de dire si un caractère a déjà été utilisé ou non. Si le caractère a été utilisé, nous avons une valeur « verif » qui passe à 1, et donc nous ne pouvons pas réutiliser cette valeur.

Si après le parcours du fichier, nous nous apercevons que la valeur n'a jamais été testé, nous allons donc créer une structure pour ce caractère de la même manière que ceux précédents, et donc re parcourir le fichier pour compter ses occurrences. Cette fonction va donc permettre de créer une liste chainée de tous les caractères du fichier.

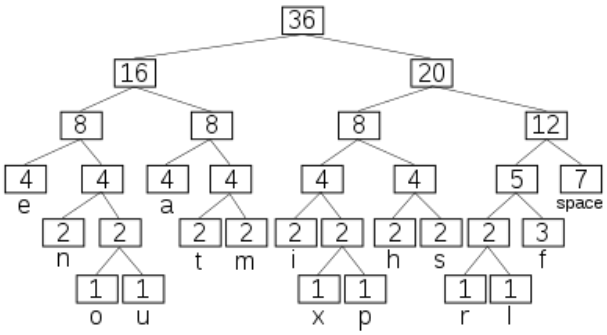
Ensuite, nous allons voir la partie la plus compliqué du projet selon nous, la deuxième étape. Il s'agit de la création de l'arbre. L'algorithme en lui-même n'est pas très long, cependant il a fallu faire attention à bien organiser les structures et les variables. A l'aide de plusieurs fonctions nous avons donc crée un arbre de Huffman, à partir d'une liste d'occurrences.

Pour commencer, nous avons directement commencé par trier notre liste, en fonction des occurrences par ordre croissant. La fonction ranger_liste retourne donc un Element*. A l'aide d'un pointeur temporaire nous allons arranger la liste de manière à la trier par ordre croissant. Nous passons un pointeur sur la liste afin de pouvoir la modifier directement sans devoir en créer une nouvelle. Ici, l'intérêt de créer un temp est de pouvoir manipuler la liste sans perdre son début, car si nous déplaçons le pointeur principal, nous avons complètement perdu le début de notre liste dans la mémoire et il est donc impossible de la retrouver.

Une fois que la liste est triée par ordre croissant en fonctions des occurrences, nous allons convertir une liste d'élément en liste de nœud, avec l'aide des structures présenter dans la première partie du rapport. Cette fonction est relativement simple, nous avons simplement à transférer la data de la première structure, dans la deuxième structure, et faire pointer les 2 pointeur left et right vers NULL, car nous n'avons actuellement pas de fils. Ces deux pointeurs serviront dans la création de l'arbre.

Ensuite, le premier pas vers la création de l'arbre huffman commence. Notre objectif ici est de créer une fonction qui retourne l'occurrence la plus petite de la liste chainée. L'objectif est de systématiquement rassembler les 2 plus faibles occurrences, dans un nœud (créer préalablement).

La fonction min_occu_LN va retourner la plus petite occurrence de la liste.



Une fois que le nœud est créé, le nœud se retrouvera à la fin de la liste. Nous allons répéter ce processus jusqu'à ce qu'il ne reste plus que 1 élément dans la liste. Le nœud de huffman possèdera uniquement la somme des 2 occurrences les plus faibles, qui seront des structures, sur son left et son right. Par exemple nous avons une liste chainée trier par ordre croissant :

2/3/5/7/8/9

Ici, nous avons nos occurrences, nous allons créer un nœud de huffman, qui va prendre en tant que right la structure qui possède 2 comme occurrence, et en tant que left la structure qui possède 3 comme occurrence. La somme des deux étant égal à 5, nous allons avoir la liste suivante :

5/7/8/9/5

Avec le dernier nœud (5) qui a : left → (structure ou occu = 2) et right → (structure ou occu = 3), de manière à la fin à obtenir un arbre de huffman ci-dessus.

Cette création successive de nœud formant un arbre huffman est possible grâce à notre dernière fonction : Arbre_Huffman. Cette fonction va donc appeler la fonction la fonction min_occu_LN deux fois, afin d'obtenir 2 structures, pour créer un nœud de huffman. Nous allons ensuite appeler 2 fonctions secondaires. La première est create_noeud_huffman, qui va simplement créer un nœud avec les structures, et la fonction add_noeud_fin_LN, qui va ajouter le nœud à la fin de la liste.

La dernière étape est la création du dictionnaire. L'objectif est de créer un dictionnaire correspondant à notre arbre de huffman. Nous devons donc faire correspondre chaque lettre de notre arbre, avec une suite de 0 et de 1. Nous savons que les lettres qui apparaissent le plus de fois, sont les plus proches de la racine, et celles qui sont les plus profond dans l'arbre sont les caractères les moins utilisés.

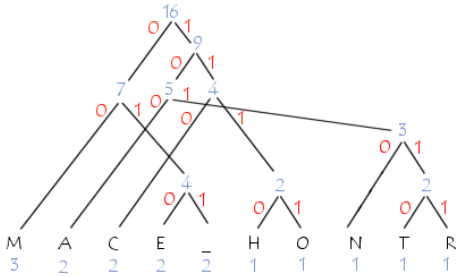
Exemple de dictionnaire :

dico.txt

```
S:0
T:10
E:110
A:111
```

Ici, le A apparait plus que le E.

Nous allons donc parcourir tout l'arbre de manière suffixe, c'est-à-dire que nous commençons par les feuilles, pour finir à la racine. Nous commençons par créer une fonction : dico. Cette fonction va utiliser la structure PileElement, pour pouvoir stocker les chemins que l'on prend pour arriver aux caractères dans l'arbre. Nous allons dépiler et empiler de manière à gagner du temps et donc de l'optimisation.



Ici par exemple, nous commençons par le M : 00, qui est donc un caractère relativement utilisé. Notre Pile aura donc 00. On dépile un 0, et on part à droite, donc en ajoute 1, et on ajoute 0 pour trouve E, Nous aurons donc dans notre pile : 010. On s'aperçoit que E est plus long que M, ce qui est normal car il a que 2 occurrences.

Ensuite, il nous reste à parcourir l'arbre et à écrire dans un fichier texte : dic.txt, les numéros attribués à chaque caractère. Ceci à l'aide notamment de la fonction écrire_dico_pile.

Nous nous servons donc d'une condition afin de savoir s'il s'agit d'un nœud de huffman ou non.

if (Arbre->left->lettre != 128), ce n'est pas un noeud de huffman si lettre != 128

De cette manière, nous empilons les chemins avec la fonction écrire_dico_pile, et dès que nous trouvons un caractère, nous écrivons dans le fichier dic.txt, ouvert en « w » :

fprintf(dic,"%c : ",Arbre->right->lettre); sans oublier le retour à la ligne fputc('\n',dic);

Une fois que nous avons écrit notre caractère dans le fichier, on remonte dans l'arbre en dépilant notre pile, et nous répétons cette action jusqu'à la racine.

Finalement, il reste l'encodage. Son rôle est de réécrire le fichier texte initial, avec le dictionnaire de suite binaire. Ça veut dire que nous devons simplement remplacer chaque caractère par sa suite binaire correspondante. Pour chaque caractère du fichier d'entrée, il va falloir chercher dans le fichier dictionnaire le code correspondant, et l'écrire dans le fichier de sortie.

Pour commencer, nous avons écrit une fonction read_char, qui retourne un char*. Son rôle est simplement de lire dans le fichier dic.txt. Nous allons donc créer un tableau dynamique afin de stocker l'information, et écrire dans ce tableau, et le retourner.

Maintenant, nous devons parcourir le fichier. Nous avons créé une fonction : dico_to_element, qui va donc parcourir le fichier dic.txt, et créer des structures de type ElementD. Le but est de re créer une liste chainée de structure, avec le caractère, écrit avec son nombre binaire associé, et son occurrence. Nous allons donc créer une structure, que l'on va retourner à la fin de la fonction. Lors de notre parcours de fichier, nous allons nous servir de la fonction read_char, qui va nous retourner un tableau avec la série binaire de notre caractère, et nous allons pouvoir le stocker dans sa structure. Nous effectuons cette fonction avec un appel récursif.

Maintenant, nous devons créer une fonction qui créer un fichier texte compressé, et le remplir à l'aide de la liste chainée compressée créer antérieurement. Nous avons créé notre fonction conversion_txt_to_huffman. Cette fonction récupère une structure, donnée en paramètre, pour l'ecrire dans un fichier encodage_huffman.txt, tant que verif est diffèrent de 1. Si notre variable verif == 0, c'est que notre structure a déjà été écrite.

Après ça, nous avons donc notre fichier encodage_huffman.txt, qui est note fichier texte compressé

3. Les fonctions secondaires

Dans la réalisation de ce projet, nous avons un fichier contenant toutes les fonctions secondaires, ce sont des fonctions simples, qui ont un seul objectif, ça peut être de l'affichage, de l'ajout d'élément à une liste ou de la création de structure.

print_tree : Cette fonction va afficher un arbre huffman, en recursif, mais uniquement les feuilles. C'est-à-dire que si tree→lettre != 128, nous n'affichons rien, sinon, on affiche la lettre et son occurrence. Un appel recursif vient ensuite pour right et left.

creer_noeud : Cette fonction créer une structure de type Nœud, avec les 2 pointeurs left et right sur NULL, car son rôle n'est pas d'ajouter un élément à une liste, et lettre et occurrence avec ce qui est donné en paramètre.

add_noeud_LN : Cette fonction ajoute une structure de type, nœud à une autre, que la fonction crée. Si une liste donnée en paramètre est égale a NULL, alors l'élément crée par la fonction va être le premier de la liste, sinon, nous ajoutons l'élément à la liste donnée en paramètre.

tree_compare : Cette fonction compare un arbre. Nous allons donc comparer 2 nœuds, et si a un moment l'enfant n'est pas égal à l'autre, alors un 0 apparait et la somme =0. Un appel récursif est utilisé pour parcourir tout l'arbre.

create__noeud_huffman : Cette fonction créer un nœud qui prend la somme des occu des 2 nœuds donnée en paramètre et qui n'a pas de lettre donc lettre = 128

suppr_noeud : va supprimer un nœud d'une élément donné en paramètre. On recherche le nœud dans la liste et on le supprime, on free le pointeur temporaire et on fait un appel récursif pour être sûr que l'élément n'est pas présent plusieurs fois.

add_noeud_fin_LN : Cette fonction va parcourir une liste chainée, des que notre pointeur temporaire arrive a temp→next NULL, c'est que l'élément actuel est le dernier. Nous allons donc ajouter un élément donné en paramètre à la fin de la liste.

afficher_LC : cette fonction affiche une listé chainée. On affiche donc le premier élément et on fait un appel récursif avec →next, si L != NULL.

créer_PileElement : Cette fonction créer une structure PileElement, et la retourne, avec un car = car donné en paramètre de la fonction

empiler : Cette fonction créer une pile si celle donné en paramètre n'existe pas, et ajoute des element à la chaine avec la fonction créer_PileElement

dépiler : Cette fonction supprime des element de la liste, et free l'élément retiré.

print_tree_H : Cette fonction affiche un arbre huffman. C'est-à-dire que nous allons uniquement afficher les caractères. Si tr→lettre != 128, on affiche, sinon, on affiche que l'occurrence (qui est donc la somme des occurrences de ses enfants). Nous finissions avec un appel récursif pour parcourir tout l'arbre.

a.Le fichier header

```
char* convert_bin(int n);
int lecture_ecriture(FILE* f1,FILE* f2);
int nbr_char_fic(FILE* fic);

---------------


Element* list_occu_char(FILE* fic);
Element* insert_list(Element* li,int lettre);
Element* ranger_liste(Element* liste);
ElementN* convert_LC_to_LCN(Element* L);
Noeud* min_occu_LN(ElementN** L_N);
void Arbre_Huffman(ElementN** L_N);

void test();
void ecrire_dico_pile(PileElement* pile,FILE* dic);
void dico(Noeud* Arbre,PileElement** nbr,FILE* dic);

void test1();
```

```
char* read_char(FILE* dic);
ElementD* dico_to_element(FILE* dic);
void convertion_txt_to_huffman(FILE* f2,FILE* f3,ElementD* L);
```

Le header sert à lister toutes les fonctionnalités de notre programme, sans montrer les détails des fonctions. Nous avons également nos structures listées dans le header.

c.Le main.c

Nous avons donc notre fichier principal, le main.c. Ce programme sert à exécuter nos fonctions et de les tester lorsqu'elles sont en cours de développement, afin de comprendre ce qui marche ou pas. Nous avons utilisé des fonctions nommées test 1, 2 et 3, qui servent à exécuter seulement quelques fonctions entre elle. Cela sert à tester notre code partiellement.

III.Conclusion générale

a.Bilan

Afin de mener à bien ce projet, nous catégorisons les fonctions dans différents fichiers en fonction de leurs domaines d'utilisation. Nous avons créé plusieurs fichiers .c, avec chacun son fichier .h dans lequel les fonctions y sont listées.

De plus, nous avons essayé de créer une structure afin de pouvoir distinguer l'arbre de Huffman de la liste des lettres, des nœuds et des codes. Par conséquent, ces structures rendent le code clair. Et donc permettent au code d'être facile à lire.

b.Les difficultés rencontrées

Il nous a fallu un certain temps pour comprendre comment créer un arbre Huffman. Une fois que vous l'avez compris, vous pouvez reprendre calmement l'écriture du code.

De plus, avant de créer plusieurs structures différentes pour distinguer l'arbre de Huffman des trois listes, nous n'en avons qu'une. Le problème est qu'il est difficile de faire la distinction entre les arbres et les différentes listes. Par conséquent, pour surmonter ce problème, nous avons créé une structure pour l'arbre Huffman et créé une structure pour chaque liste créée.

c.Un projet enrichissant

Ce projet nous a permis d'entrer dans le domaine de l'informatique professionnelle car nous nous sommes occupés de la compression de fichiers texte, quelque chose d'encore utilisé par tout le monde très souvent. De plus, ce projet nous a permis de mettre en application nos compétences en programmation et en algorithmie, car ça nous a permis de manipuler des structures des arbres et des listes chainées.

d.Quels sont les autres compressions possibles ?

Dans ce projet, nous avons réalisé une compression huffman, qui est une compression de fichier texte. Cependant il est possible de compresser des images. La compression d'image est une application de compression de données sur des images numériques. Cette compression a pour effet de réduire la redondance des données d'image, de sorte qu'elles peuvent être stockées sans prendre beaucoup de place et les transmettre rapidement. Il serait enrichissant d'étudier et de comprendre le fonctionnement de cette méthode de compression, car elle joue un rôle essentiel lors du partage de documents ou de fichiers volumineux.