Université Paris Scalay

Faculté des Sciences

# Algorithme de compression et décompression de texte

PROJET IPF

*Présenté par :*

*Wang Marc — Pourchot Alistair*

Groupe : LDDIM2

# Introduction

**L**a compression de données en informatique consiste à transformer une suite de bits en une suite plus courte, de telle sorte que l'on puisse retrouver la suite d'origine en faisant une décompression. Cela pemet de réduire grandement la taille d'un fichier ; par exemple, les dossiers compréssés en .zip sont les plus utilisés et lorsqu'on les décompresse on obtient bien les fichiers de base, ce qui évite de télécharger directement des gros fichiers depuis internet par exemple.

Il existe également des algorithmes de compression qui autorisent la perte d'information, mineure ou majeure, à condition qu'elle ne soit pas visible par l'oeil humain. Par exemple si on regarde une vidéo et qu'un pixel est oublié pendant une fraction de seconde, on n'en tient pas compte.

On s'intéressera ici seulement à la compression sans perte, en particulier en suivant le modèle de Huffman.

# Contents

# 1 Types de données

## 1.1 File de priorité

Nous avons conçu une file de priorité afin d'identifier les occurrences les plus élevées des mots, dans le but d'assurer une compression efficace. Pour ce faire, nous utilisons une structure qui stocke l'occurrence de chaque caractère ainsi que l'arbre correspondant, et nous appliquons l'algorithme de Huffman.

```
val cree_collec : int array -> (int * tree) list
(* Convertit un tableau d'entiers d'indice i (le code ASCII) et qui à son indice contient son occurence en une collection*)
```

Figure 1: file de priorité

```
int array -> (int * tree) list
let cree_collec tab =
  let a = 0 in
  let res = [] in
  let rec loop tab1 tab2 i =
    if i >= Array.length tab then tab2
    else
    if tab1.(i) == 0 then loop tab1 tab2 (i +1)
    else loop tab1 (add (tab1.(i), Leaf(i)) tab2) (i +1)
  in loop tab res a
```

Figure 2: création de la collection

## 1.2 Structures auxilières

Puisque l'on travaille sur des fichiers qui ne sont pas en binaire mais en ASCII, il est fondamental d'avoir de quoi convertir de l'un vers l'autre et vice-versa, c'est pour cela que l'on dispose d'un nouveau type : (int*string) list, ou l'entier est le code en ASCII et le binaire est en string.

```
tree -> (int * string) list
let convert arbre =
  let i = "" in
  let res = [] in
  let rec loop arbre i res =
    match arbre with
    |Leaf b -> (b, i) :: res
    |Node (c, d) ->loop c (i^"0") res @ loop d (i^"1") res
  in
  loop arbre i res
```

Figure 3: type aux

```
Code ASCII : 102 et Code binaire : 000
Code ASCII : 110 et Code binaire : 001
Code ASCII : 97 et Code binaire : 01
Code ASCII : 115 et Code binaire : 10
Code ASCII : 105 et Code binaire : 110
Code ASCII : 116 et Code binaire : 111
```

Figure 4: exemple sur satisfaisant

# 2 Code

## 2.1 Répartition du code dans les fichiers

Pour ce qui est de la répartition du code, nous n'avons utilisé seulement les fichiers de base, sans en créer de nouveaux. Le code a été reparti dans plusieurs fichier selon leur utilité. Par exemple, dans le fichier heap.ml nous avons construit notre structure de données utilisée pour la décompression, à savoir des arbres binaires valués aux feuilles.

```
1 type tree =
2 | Leaf of int
3 | Node of tree * tree
4
5 type file  = int * tree list
```

Pour utiliser ces arbres, nous avons codés un ensemble de fonctions dans ce-dit fichier, en particulier des fonctions de conversion, par exemple pour convertir nos arbres en tableaux appelés 'collection'.

Heap.ml dispose de tout ce dont nous avons besoin pour notre structure de compression.

Ensuite, dans notre fichier huffman.ml nous avons les fonctions les plus compliquées à réaliser, notamment compression, décompression ainsi que leur fonctions auxiliaires.

Dans nos fonctions de compression et de décompression, nous avons choisi de garder le même nom de fichiers après la compression, en rajoutant l'extension ".hf" à la place de ".txt", puis nous rajoutant le suffixe "_decomp" une fois décompressé, en remettant à nouveau notre extension de départ ".txt".

Enfin, notre dernier fichier comportant du code est le fichier comportant le code principal : "huff.ml". Dans celui-ci se trouvent les manières dont notre code doit se comporter selon la demande de l'utilisateur. En l'occurence, on dispose de 4 comportements différents, tous obtenables directement en effectuant ./huff.exe –help [1] dans notre terminal après avoir compilé. On obtient ceci :

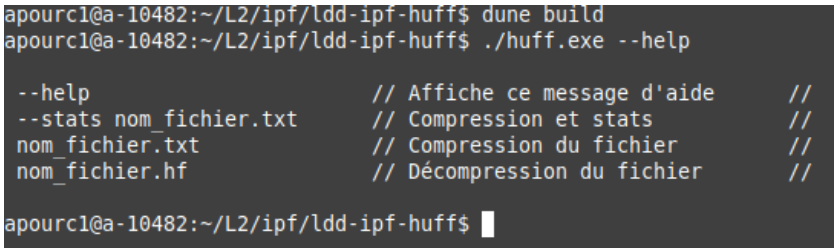


Figure 5: capture d'écran de l'execution avec –help

On peut remarquer qu'une des commande consiste à la compression et à l'affichage des statistiques du fichier, essayons sur notre fichier produit.txt contenant du texte parlant du lorem ipsum.

[1] En fait, on voulait trouver un moyen de pouvoir executer juste en écrivant huff –help (donc sans le .exe), mais on a pas trouvé de manière de le faire. On a ensuite essayé une autre maniere qui consitait a executer puis a attendre les instructions de l'utilisateur (ce code est dailleurs toujours la en commentaite dans huff.ml) On se contentera donc de l'éxecuter comme ça

```
apourc1@a-10482:~/L2/ipf/ldd-ipf-huff$ dune build && ./huff.exe --stats produit.txt
Voici les stats du fichier : produit.txt
(5,x) (52,v) (187,u) (174,t) (171,s) (108,r) (15,q) (51,p) (92,o) (112,n) (87,m) (129,l)
(3,j) (221,i) (8,h) (27,g) (15,f) (236,e) (60,d) (86,c) (16,b) (169,a) (2,V) (3,U) (5,S)
(1,Q) (5,P) (5,N) (3,M) (1,L) (5,I) (2,F) (2,E) (3,D) (2,C) (2,A) (41,.) (37,,) (385, ) (
1,\n)
```

Figure 6: execution de stats sur du lorem ipsum

On voit alors que les statistiques en question sont le nombre d'occurence de chaques caractères. (Notez que le "\n" à la fin correspond au saut de ligne).

## 2.2 Du fil à retordre

Lors de ce projet, plusieurs questionnement nous sont parvenus. Comment stocker de manière concise nos arbres de Huffman dans nos fichiers compressés, et également comment lire nos fichiers compressés de telle sorte que l'on puisse restituer notre fichier original.

On a alors essayé plusieurs manières différentes : au début, on avait une idée qui n'utilisait pas sérialisation et désérialisation. On n'utilisait pas non plus les fonctions de Bs, mais on écrivait les 0 et les 1 en string, donc chaque 0 et chaque 1 constituait un octet, ce qui ne compressait pas du tout, car la taille du fichier obtenu était souvent plus grande que celle de base. Finalement, après nombreuses remodifications de notre code, on a parvenu à utiliser les fonctions de Bs pour écrire des bits (et non des octets), ce qui, en effet, réduisait la taille de notre fichier une fois compressé.

## 2.3 Tests

Enfin, pour vérifier que nos fichiers étaient bien compressés, on a effectués plusieurs tests. Tout d'abord le premier test avec satisfaisant :

```
Taille fichier avant compression : 12 octets
Taille fichier après compression : 13 octets
```

Figure 7: compression sur satisfaisant

n'est pas très concluant. En effet, le fichier est déjà petit de base, et lors de la compression on doit rajouter notre arbre de Huffman, ce qui prend à priori plus de place que le mot "satisfaisant".

Nous avons également testé notre compression, à l'inverse, sur un fichier très long de 2M5 caractères (long.txt), constitué de lorem ipsum copié collé de nombreuses fois)

```
Taille fichier avant compression : 2533310 octets

Taille fichier après compression : 1448046 octets
```

Figure 8: compression sur long.txt

Enfin, nous avons testé notre compression sur un fichier de texte contenant des caractères spéciaux et des symboles (speciaux.txt) :

```
Taille fichier avant compression : 7544 octets

Taille fichier après compression : 5562 octets
```

Figure 9: compression sur speciaux.txt

# 3 Conclusion

## 3.1 Répartition du travail

Cela fait plusieurs projets déjà que nous travaillons ensemble, et nous formons un assez bon groupe. Lors de ce projet, nous étions toujours côte à côte pour travailler, afin de pouvoir s'entraider et de pouvoir communiquer assez facilement sur nos intentions[2]. Ici, Marc a réalisé le fichier heap.ml, et Alistair s'est occupé de ligne de commande, et nous avons travaillé ensemble sur Huffman.ml. Notre groupe a été très complémentaire dans la réalisation du projet, et nous avons surmonter nos doutes et nos questionnements ensemble.

## 3.2 Fin

Enfin, ce projet nous a été, pour nous deux, une belle experience. Tout d'abord, nous a permis de découvrir en quoi consistait l'algorithme de Huffman, mais également de voir comment un fichier pouvait être compressé sans perdre d'informations. Cette réalisation a été très interessante et nous a permis encore mieux d'apprendre à travailler ensemble.

[2]sauf pendant les vacances de Noel, chacun respectivement sur son pc mais toujours en communication avec l'autre sur les modifications, les problèmes, etc