Rapport du Projet Compression de texte

avec l'algorithme Huffman

Maram Sall GUEYE Reina AL MASRI

1 Introduction générale et organisation du projet.

Ce projet de compression de texte a été réalisé dans le cadre de l'UE Introduction à la Programmation Fonctionnelle (DLIN231), l'objectif étant d'implémenter un algorithme de compression et de décompression basé sur le codage de Huffman (David A. Huffman, 1925-1999, informaticien Américain).

Organisation : l'une de nous a travaillé sur la fonction **`compress`**, et l'autre sur la fonction **`decompress.`** L'une de nous a plutôt travaillé avant les vacances, et l'autre après. Pour toutes les autres fonctions, nous avons collaboré en partageant nos idées et en codant ensemble certains jours. Nous avons également utilisé FramaGit pour centraliser le code, suivre les versions et faciliter le partage de notre travail.

2 Description des fichiers

Voici les fichiers présents dans la version finale du projet :

- **`bs.mli`** : fichier contenant les interfaces pour la gestion des flux binaires (fourni par le prof)
- **`bs.ml`** : fichier implémentant les fonctions pour lire et écrire des bits dans des fichiers binaires (fourni par le prof)
- **`heap.mli`** : fichier contenant les interfaces des fonctions pour la gestion et manipulation d'un tas (heap).
- **`heap.ml`** : fichier implémentant les fonctions pour la gestion et manipulation d'un tas (heap), utilisé pour construire l'arbre de Huffman.
- **`huffman.ml`** : fichier implémentant les fonctions principales pour la compression (**`compress`**) et la décompression (**`decompress`**) des fichiers.
- **`huff.ml`** : regroupe les fonctions principales pour l'exécution du programme, notamment le traitement des arguments en ligne de commande.
- **`test.ml`** : fichier comportant une partie des tests realises pour heap et huffman
- **`Le_parfum_Suskind.txt`** : roman de plus de 55 000 octets utilisé pour tester la compression et la décompression et qui nous a notamment forcé à utiliser un Buffer lors de la décompression.

3 Fonctions

La version actuelle de heap.ml est une version améliorée qui nous permet une insertion et une suppression facile pour des ensembles plus grands et des opérations intensives. En effet, bien que les listes triées ait pu présenter une simplicité d'implémentation, l'utilisation des tas binaires sur des tableaux représente pour nous une efficacité globalement plus satisfaisante et beaucoup plus appropriée pour de grands fichiers.

3.1 Fichier huffman.ml

3.1.1 Fonctions pour compress

- **`input_code : in_channel -> int`**
  Fonction qui lit un octet d'un fichier ouvert en tant qu'**`in_channel`** et renvoie son code ASCII. Renvoie **`-1`** si la fin du fichier est atteinte.
- **`char_freq : in_channel -> int array`**
  Fonction qui calcule la fréquence de chaque caractère dans un fichier en remplissant un tableau de taille 256 (ASCII) où chaque index correspond à un caractère.
- **`freq_heap : int array -> (int * heap) heap.heap`**
  Fonction qui convertit un tableau de fréquences en un tas min (heap), où chaque élément est une paire (fréquence, feuille de Huffman).
- **`build_huff_tree : (int * tree) heap.heap -> tree`**
  Construit un arbre de Huffman à partir d'un tas de fréquences. Combiner récursivement les deux nœuds de plus basse fréquence jusqu'à obtenir un arbre unique.
- **`print_tree : tree -> string -> unit`**
  Fonction qui affiche l'arbre de Huffman sous forme hiérarchique pour faciliter les tests et la visualisation. Le second paramètre est utilisé pour l'indentation.
- **`code_of_tree : tree -> (char * string) list`**
  Fonction qui associe chaque caractère de l'arbre de Huffman à son code binaire, représenté sous forme de chaînes.
- **`serialize_tree : Bs.ostream -> tree -> unit`**
  Fonction qui sérialise l'arbre de Huffman en écrivant sa structure dans un flux binaire pour pouvoir le sauvegarder dans un fichier compressé.
- **`write_data : in_channel -> (char * string) list -> Bs.ostream -> unit`**
  Fonction qui écrit les données compressées dans un fichier en remplaçant chaque caractère par son code binaire en fonction de l'arbre de Huffman.
- **`compress : string -> unit`**
  Fonction principale de compression. Lit un fichier, génère son arbre de Huffman, compresse ses données et les enregistre dans un fichier avec l'extension **`.hf`**.

En combinant toutes ces fonctions, on obtient une fonction **`compress`** ainsi définie :

```
let compress f =
  let in_c = open_in f in
  let freq_tab = char_freq in_c in
  let freq_heap = freq_heap freq_tab in
  let huff_tree = build_huff_tree freq_heap in
```

```
let char_codes = code_of_tree huff_tree in
let f2 = f ^ ".hf" in
let cout = open_out f2 in
let os = Bs.of_out_channel cout in
serialize_tree os huff_tree;
seek_in in_c 0;
write_data in_c char_codes os;
Bs.finalize os;
close_in in_c;
close_out cout
```

L'explication de l'algorithme de Huffman et les fonctions de sérialisation ont été faites dans les supports fournis pour le projet (cf. le sujet du projet et la feuille de séance 3).

3.1.2 Fonctions pour la ligne de commande

- **`stats : string -> unit`**
  Affiche des statistiques de compression, telles que la taille initiale et compressée, ainsi que le taux de compression en pourcentage.
- **`help : unit -> unit`**
  Affiche un message d'aide dans la console, listant les options disponibles pour le programme.

3.1.3 Fonctions pour `decompress`

- **`deserialize_tree : Bs.istream -> tree`**
  Fonction qui reconstruit un arbre de Huffman à partir d'un flux binaire sérialisé.
- **`decompress : string -> unit`**
  Fonction principale de décompression. Lit un fichier compressé, reconstruit l'arbre de Huffman, décode les données compressées, et les enregistre dans un nouveau fichier sans l'extension **`.hf`**.

3.2 Fichier heap.ml

La documentation ainsi que les types se trouvent dans le fichier **`heap.mli`** . On se propose d'expliquer les fonctions **`add`** et **`remove_min`** pour le type de donnée des tas min.

3.2.1 Qu'est-ce qu'un tas (heap) ?

Un tas (heap) est une structure de données binaire où chaque nœud parent respecte une propriété spécifique par rapport à ses enfants. Dans un tas minimal (ce qui est la structure de données qu'on a utilisé pour l'algorithme de Huffman), chaque parent a une valeur inférieure ou égale à celle de ses enfants (cf. UE Algorithmique Avancée, OLIN235B)
On a décidé de représenter cette structure de donnée sous ocaml avec un tableau : l'élément à l'indice **`i`** a :

- un parent : d'indice **`(i - 1) / 2`**
- un enfant gauche : d'indice **`2 * i + 1`**
- un enfant droit : d'indice **`2 * i + 2`**

Exemple d'un tas minimal :

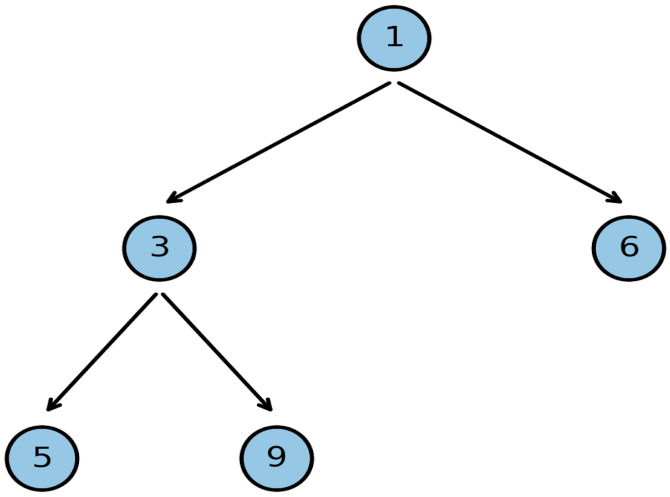


Tableau correspondant : `[1, 3, 6, 5, 9]`
Dans notre cas, on avait pas des entiers, mais plutôt des paires `int * tree.`

3.2.2 Fonction `add` :

Déroulement : Remonter l'élément ajouté (à l'indice `i`) jusqu'à ce qu'il respecte la propriété du tas (élément parent ≤ éléments enfants).

Voici le code :

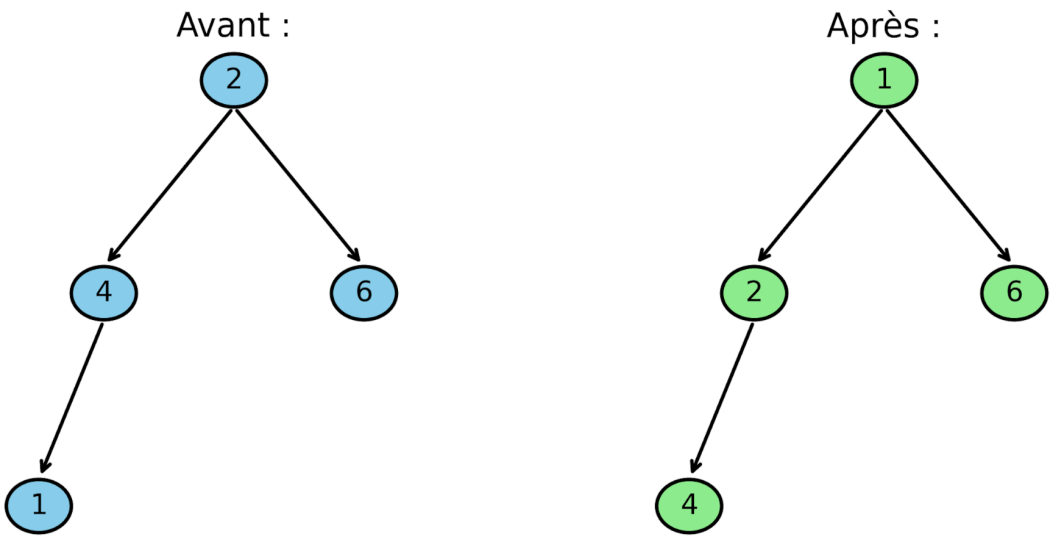
```
let add x h =
  let h' = Array.append h [| x |] in
  remonte_heap h' (Array.length h' - 1);
  h'
```

Exemple :

- Initial : `h = [(2, Leaf 'a'); (4, Leaf 'b'); (6, Leaf 'c')]`
- Ajout : `x = (1, Leaf 'd')`
- Étapes :
  - Ajouter `(1, Leaf 'd')` à la fin :
    `h' = [(2, Leaf 'a'); (4, Leaf 'b'); (6, Leaf 'c'); (1, Leaf 'd')]`
  - Remonter `(1, Leaf 'd')` pour être à la bonne position :

On a : `h' = [(1, Leaf 'd'); (2, Leaf 'a'); (6, Leaf 'c'); (4, Leaf 'b')]`

Schéma illustratif :



3.2.3 Fonction `remove_min`

L'objectif est de retirer le plus petit élément (racine) du tas tout en maintenant la propriété du tas minimal.

Déroulement :

1. Remplacer la racine (indice `0`) par le dernier élément du tableau.
2. Réduire la taille du tableau de 1.
3. Faire descendre l'élément de la racine pour restaurer la propriété du tas.

3.2.4 Résumé

Fonction	Rôle	Complexité
`add`	Ajoute un élément en maintenant l'ordre	O(log n)
`remove_min`	Retire la racine et rétablit l'ordre	O(log n)

Ces fonctions garantissent que le tas minimal reste conforme après chaque modification.

4 Tests et utilisation

4.1 Réalisation des tests

4.1.1 Tests sur des petits fichiers :

Méthodologie initiale : Au début du projet, les tests ont été réalisés avec un fichier contenant le mot "satisfaisant". Pour valider les résultats, nous avons utilisé des affichages intermédiaires à différentes étapes (construction de l'arbre, génération des codes, compression, etc.).

Observations : Avec des fichiers aussi petits, la taille du fichier compressé était souvent plus grande que l'original, car il inclut l'arbre de Huffman et les données elles-mêmes. Notamment, en testant la fonction compress, on a créé un fichier contenant ‘satisfaisantsatisfaisantsatisfaisant…’ plusieurs fois. On a obtenu une suite binaire qui, si visualisée, nous donne des caractères différents au début, puis des caractères qui se répètent. Cela nous a donné l’indication qu’on est bien sur la bonne voie.

Voici un exemple sur un affichage fait dans le terminal lors des premiers tests :

```
Char a : 3 occurrences
Char f : 1 occurrence
Char i : 2 occurrences
Char n : 1 occurrence
Char s : 3 occurrences
Char t : 2 occurrences
Arbre de Huffman :
Node
  Node
    Node
      Leaf 'n'
      Leaf 'f'
    Leaf 's'
  Node
    Leaf 'a'
    Node
      Leaf 't'
      Leaf 'i'

Code de chaque char :
Char i : 111
Char t : 110
Char a : 10
Char s : 01
Char f : 001
Char n : 000
```

4.1.2 Tests sur des gros fichiers :

Méthodologie avancée : Une fois le programme stabilisé, nous avons testé les performances avec de grands fichiers texte. Nous avons téléchargé des livres numériques depuis le site Project Gutenberg proposé.

Objectif : Vérifier que la compression est efficace sur des données réalistes, en mesurant la taille des fichiers avant et après compression, ainsi que le temps d'exécution. Ces tests ont également permis de s'assurer que les structures de données utilisées (tas, arbre de Huffman) gèrent correctement les grandes quantités d'informations.

Voici un exemple du déroulement de la compression du fichier **`Le_parfum_Suskind.txt`** :

commande : **`./huff --stats Le_parfum_Suskind.txt`**

```
Compression en cours pour le fichier : Le_parfum_Suskind.txt
Compression terminée.
Statistiques de compression :
Taille originale : 553872 octets
Taille compressée : 316152 octets
Taux de compression : 42.92%
```

4.3 Lignes de commandes

Pour compiler, on a pu utiliser la commande **`dune`** (fichiers fourni). Alternativement, on pourra utiliser la commande : **`ocamlc -o huff heap.ml bs.ml huffman.ml huff.ml`**
Voici les commandes pour exécuter les fichiers :

`./huff --help :` Affiche un message d'aide qui présente les différentes options de notre programme

`./huff <fichier> :` Crée une version compressée de fichier dans fichier.hf

`./huff --stats <fichier> :` Crée une version compressée de fichier dans fichier.hf en affichant les statistiques de compression

`./huff <fichier>.hf :` Crée une version décompressée de fichier.hf dans fichier

5 Conclusion, remarques, améliorations possibles

Notre projet a ainsi été réalisé selon une approche qui, tout en proposant une solution par étapes, nous a incité à chercher à améliorer nos résultats, n'hésitant pas à revenir sur nos choix précédents pour optimiser notre travail.

Améliorations possibles: Dans certain cas, il nous ait possible de compresser un fichier déjà une fois compressé, et cette 2e compression a un taux de 1.24%, ce qui nous pousse à croire que la 1ere compression bien que très efficace (taux > 40%) pourrait parfois être plus optimale.

Ce projet a été une belle opportunité d'apprentissage.