TP #27 — Compression

Sujet inspiré de Jean-Baptiste Bianquis.

Le but de ce sujet est d'implémenter l'algorithme « de Huffman » en OCaml, sous la forme d'un petit programme exécutable permettant de (dé)compresser n'importe quel fichier.
Les premières parties de ce sujet sont à réaliser de façon « classique », typiquement en appelant les fonctions développées depuis un interpréteur interactif (idéalement `utop`). Un certain nombre de ces fonctions interagissent néanmoins avec votre environnement d'exécution en créant ou lisant des fichiers (texte ou autre).

Fichier test

1. Décidez d'un fichier test pour votre programme. Celui-ci peut être n'importe quoi (il n'a pas besoin d'être un fichier texte), mais un fichier trop court ou déjà compressé risque de ne pas donner de résultats satisfaisants. Le fichier utilisé en guise d'exemple dans ce sujet sera cette version des *Adventures of Huckleberry Finn*.

Lecture/écriture de bits

Le principe même de la compression « de Huffman » consiste à coder des symboles par un nombre variable de bits. Il est donc nécessaire pour une implémentation d'être capable d'écrire & lire des bits individuels dans un fichier, ce qui n'est pas une opération de base des systèmes de fichier modernes, dont l'unité élémentaire est l'octet (huit bits).
On fournit ici des implémentations de toutes les fonctions nécessaires pour ces lectures & écritures, qui suivent le principe suivant : chaque fois que l'on a (disons) écrit 8 bits dans un flux « bit », on écrit un octet les représentant dans un flux « octet » associé. Un flux « bit » est alors représenté par un état contenant le flux « octet » associé et deux variables contenant les éventuels bits non encore écrits dans celui-ci, ainsi que leur nombre (entre 0 et 7). Une fois le dernier bit écrit dans le flux « bit », on écrit (s'il ne l'a pas déjà été) l'octet contentant les derniers bits écrits, et un dernier octet indiquant combien de ses bits (entre 0 et 8) sont *significatifs* (les autres étant des bits de « padding »).
Ces flux bits sont représentés par les types :

```
type io_chan = In of in_channel | Out of out_channel

type bitchannel = {
    mutable acc : int ;
```

```
    mutable cnt : int ;
    mutable len : int ;
    chn : io_chan
}
```

Les fonctions fournies sont :

```
bc_open_in     : string -> bitchannel
bc_plug_in     : in_channel -> int -> bitchannel
bc_open_out    : string -> bitchannel
bc_plug_out    : out_channel -> bitchannel
bc_write_flush : bitchannel -> unit
bc_close       : bitchannel -> unit
bc_write_bit   : bitchannel -> int -> unit
bc_read_bit    : bitchannel -> int
```

2. Lisez et comprenez le code des fonctions :
   - `bc_open_in`
   - `bc_open_out`
   - `bc_write_flus`
   - `bc_close`
   - `bc_write_bit`
3. Écrivez une ou des fonctions de test permettant de vérifier le bon fonctionnement des fonctions :
   - `bc_open_in`
   - `bc_open_out`
   - `bc_close`
   - `bc_write_bit`
   - `bc_read_bit`

   On suggère pour cela notamment d'écrire une fonction qui écrit une suite de bit (de longueur variable) passée en argument dans un flux « bit » associé à un fichier, puis lit les bits depuis ce fichier et compare le résultat avec la suite initiale.
4. *(Bonus.)* Je ne suis pas satisfait de ma fonction `bc_read_bit`. Vous aurez peut-être mieux à proposer ?

Table de fréquence

5. Écrivez une fonction OCaml :
   `build_freq_table : string -> float array`
   telle que `build_freq_table path` lit le fichier de chemin `path` et s'évalue en `float array` de longueur 256 qui à l'indice i contient la fréquence d'apparition de l'octet i dans le fichier.

Pour le fichier test, vous devriez par exemple trouver une fréquence ≈ 0.168 pour 32, et ≈ 0.079 pour 101.
Remarque. Utilisez la fonction `input_byte` pour la lecture d'un octet.

Construction de l'arbre & tableau de compression

On définit le :
```
type comp_tree = L of int | N of comp_tree * comp_tree
```
pour la représentation de l'arbre de compression.

6. Implémentez une structure de file de priorité impérative « min » pour des éléments dont les priorités sont représentées par des `float` et les valeurs par des `comp_tree`. Vous pouvez ou bien adapter votre implémentation à base de tas binaire d'un précédent TP, ou bien suivre une approche par *skew heaps* comme à un précédent DS, ou bien utiliser un ABR (auto-équilibré ou non), ou bien implémenter une version naïve à base de `list` (la performance de la file de priorité n'a pas un grand impact ici ; faites au plus rapide).
   Dans tous les cas, vous devez aboutir à des fonctions :
   ```
   pq_init : float array -> pq
   pop_min : pq -> comp_tree * float
   push    : pq -> comp_tree * float -> unit
   ```
   qui permettent respectivement d'initialiser une file de priorité de type `pq` avec un tableau de fréquence, extraire un élément de valeur minimale, et ajouter un élément à la file.
   En particulier, un appel à `pq_init ft` devra construire une file de priorité dont les éléments sont de la forme `L i, f` avec `ft.(i) = f`, pour tous les éléments `i` de `ft` de fréquence non nulle.
7. Testez.
8. Écrivez une fonction OCaml :
   ```
   build_comp_tree : pq -> comp_tree
   ```
   qui implémente l'algorithme « de Huffman » pour construire un arbre de compression à partir de la file de priorité donnée en argument.
9. Écrivez une fonction OCaml :
   ```
   build_comp_table : comp_tree -> int list option array
   ```
   qui construit et s'évalue en un tableau de longueur 256 dont l'élément i est égal à `Some c` si i apparaît dans l'arbre de compression argument de la fonction avec un code associé `c` (encodant comme une `int list` le chemin de la racine de l'arbre à la feuille de valeur i), et `None` sinon.
10. Testez. Pour le fichier test du sujet, vous devriez par exemple obtenir des codes de 3, 4, 5, 6, 7 bits pour les symboles 32, 101, 105, 103, 107 respectivement.

En fonction de votre implémentation de la fonction précédente, il se peut que les codes soient représentés « à l'envers », c'est à dire qu'ils sont préfixes dans une

lecture de la fin de la liste vers le début. Pour éviter un calcul de renversement de liste à chaque écriture (ou lecture, au choix), il peut être pratique de calculer une nouvelle table où les code sont stockés à l'endroit.

11. Écrivez éventuellement une fonction OCaml :

```
rev_comp_table : int list option array
                 -> int list option array
```

qui construit une nouvelle table de compression pour les codes stockés à l'endroit.

12. Écrivez une fonction OCaml :

```
build_decomp_table : int list option array
                     -> (int list, int) Hashtbl.t
```

qui construit la table de décompression associée à son argument (qui contient une association `(c, i)` exactement quand ce dernier contient une valeur `Some c` à l'indice `i`).

Écriture & lecture d'un fichier compressé

13. Écrivez une fonction OCaml :

```
write_comp_symb : int list option array -> bitchannel
                  -> int -> unit
```

telle que `write_comp_symb ctbl chan symb` écrit le code associé par `ctbl` au symbole `symb` sur le flux « bit » *déjà ouvert* représenté par `chan`.

14. De même pour une fonction OCaml :

```
read_comp_symb : (int list , int) Hasthbl.t
                 -> bitchannel -> int
```

15. Testez.

16. Écrivez une fonction OCaml :

```
huff_compress : string -> string -> unit
```

telle que `huff_compress ipath opath` lit un fichier au chemin `ipath` et en écrit une version compressée par l'algorithme « de Huffman » tel qu'implémenté ci-dessus dans un nouveau fichier au chemin `opath`.
Pour le fichier test du sujet, vous devriez obtenir un fichier de taille 366 419 octets (au lieu des 622 463 originaux).

17. Écrivez une fonction OCaml :

```
huff_decompress : string -> string
                  -> (int, int) Hasthbl.t -> unit
```

telle que `huff_decompress ipath opath dctbl` lit un fichier de chemin `ipath` compressé avec la table de compression associée à `dctbl` et écrit sa version décompressée dans un nouveau fichier de chemin `opath`.
Remarque. Utilisez la fonction `output_byte` pour l'écriture d'un octet.

18. Testez.

Sérialisation

Comme montré par l'écriture de la fonction `huff_decompress`, la décompression nécessite la connaissance des symboles associés à chaque code, et l'on ne peut pas déduire cette information de la seule donnée d'un fichier compressé.
Pour rendre la décompression plus aisée d'utilisation, nous allons préfixer chaque fichier compressé de l'arbre de compression utilisé pour sa compression. Un fichier binaire permettant seulement un stockage linéaire, nous allons d'abord devoir *sérialiser* l'arbre. Plus précisément, on souhaite avoir deux fonctions :

```
serialize_tree   : comp_tree -> int list
unserialize_tree : int list -> comp_tree
```

inverses l'une de l'autre.
Elles sont en fait relativement simples à écrire de façon raisonnablement efficace : pour sérialiser un arbre, on effectue simplement un parcours qui sérialise une feuille `L v` en `[1 ; v]` et un nœud interne `N (t1, t2)` en `[0] @ st1 @ st2` avec `st1` et `st2` les sérialisations de `t1` et `t2`.

19. Écrivez la fonction `serialize_tree` implémentant cette sérialisation.
20. De même pour la fonction `unserialize_tree` qui inverse celle-ci.
21. Testez.

Un fichier compressé est maintenant donné par la sérialisation de la paire `sct, cf`, où `sct` est la sérialisation d'un arbre de compression comme obtenue ci-dessus, et `cf` la suite d'octets encodant le fichier compressé comme calculé à la partie précédente. Pour implémenter cette sérialisation, on propose simplement d'écrire `sct` octet par octet dans l'ordre de lecture, en faisant précéder cela de sa taille encodée sur deux octets (ce qui doit largement suffire).

22. Écrivez une fonction OCaml :
`huff_compress_wt : string -> string -> unit`
qui procède comme décrit ci-dessus. Il pourra être utile (mais pas nécessaire) d'utiliser la fonction `bc_plug_out` (fournie) afin de créer un flux d'écriture « bit » à partir d'un flux d'écriture « octet » déjà existant.
23. Écrivez une fonction OCaml :
`huff_decompress_wt : string -> string -> unit`
qui décompresse un fichier compressé par la fonction précédente.
24. Testez

Programme indépendant

On souhaite maintenant produire un programme exécutable indépendant qui permet depuis un terminal de compresser ou décompresser un fichier avec les fonctions écrites ci-dessus.

25. Écrivez une fonction OCaml :

`main : unit -> unit`

qui utilise `Sys.argv` pour compresser ou décompresser un fichier en fonction des arguments fournis à l'exécutable.

26. Testez, en compilant votre programme complet avec `ocamlopt`.
Par exemple :

```
> ./hufc huck_finn.txt huck_finn.huf
> ./hufd huck_finn.huf huck_finn2.txt
> diff -q huck_finn.txt huck_finn2.txt
```