TRAVAUX PRATIQUES XXVI

Algorithme de Huffman

L'objectif de ce TP est d'implémenter l'algorithme de Huffman en OCaml et de l'utiliser pour la compression et décompression de message.

A Compression et décompression

Dans cette section, on suppose donné l'arbre binaire du code utilisé et on se contente de l'utiliser. On utilise le code binaire optimal construit à partir des fréquences d'apparition des lettres dans la langue française, dont l'arbre est donné dans le fichier `francais.ml` (variable `arbre_francais`).

A.1 Généralités

On a vu en cours qu'un arbre de code binaire optimal (qui contient au moins 2 lettres) est nécessairement un arbre strict. On utilise donc le type d'arbres suivant dans ce TP pour représenter les arbres de code :

```
type arbre_code =
    | F of char
    | N of arbre_code * arbre_code
```

1. Prenez le temps de lire et comprendre le fichier `francais.ml`. La fonction `affiche_arbre` vous servira à tester vos fonctions en vous donnant la possibilité d'afficher les arbres renvoyés par vos fonctions (on ne travaillera pas dans utop).
2. Créer un fichier `compression.ml` qui contient les lignes de code suivantes :

   comp.ml
   ```
   open Francais

   let () = affiche_arbre arbre_francais; print_newline ()
   ```

   la ligne `open Francais` sert à ouvrir le module défini par le fichier `francais.ml` (et donc accéder à ses variables et fonctions sans avoir à rajouter `Francais.` au début). La ligne suivante est un test pour vérifier que la compilation fonctionne.
3. Compilez votre fichier `compression.ml` (ainsi que sa dépendance `francais.ml`) avec la ligne de commande `ocamlopt francais.ml compression.ml`. Attention, l'ordre dans lequel vous donnez les fichiers au compilateur est important en OCaml.
4. En lisant attentivement l'arbre du code binaire des lettres dans la langue Française, donnez le code binaire de la lettre 'a' et le code binaire du caractère ' ' (espace).

Type string (chaines de caractères)

Une chaîne de caractères en OCaml se manipule un peu comme un tableau immuable de caractères. Si `m` est de type `string`, alors on accède au caractère d'indice i dans `m` par la syntaxe `m.[i]`. La fonction `String.length` renvoie la longueur d'une chaîne de caractères (nombre de caractères). La fonction `String.make : int -> char -> string` permet de créer une chaîne de caractère de la même façon qu'on initialiserait un tableau de caractères : `String.make n 'a'` crée la chaîne constituée de n 'a' : "aa...a".

On peut concaténer deux chaînes de caractères avec l'opération `^` : `m1 ^ m2` est la concaténation des chaînes de caractère `m1` et `m2`. On n'hésitera pas dans ce TP à faire des concaténation d'une chaîne de caractère avec un caractère, pour la rallonger.

Astuce pour les plus à l'aise d'entre vous : Il existe une fonction `String.concat : string -> string list -> string` telle que `concat sep ss` concatène la liste `ss` de chaînes de caractères, en insérant la chaîne de caractères séparatrice `sep` entre chacune d'entre elles. Sa complexité est linéaire en la taille totale de la chaîne obtenue, c'est donc une bien meilleure façon de construire une chaîne qu'en ajoutant les caractères à la fin un par un (complexité quadratique, car chaque ajout de lettre est en complexité linéaire en la taille de la chaîne obtenue).

Commençons par coder quelques petites fonctions pour prendre la main sur la manipulation du type string :

5. Écrire une fonction `mirroir : string -> string` qui renvoie la chaîne de caractère qui correspond à celle donnée en entrée, mais lue à l'envers. Par exemple, `mirroir "mot"` doit renvoyer la chaîne "tom".
   Indication : on utilisera length et make, et on lira chaque caractère dans le bon ordre.

Nous sommes maintenant prêts à s'intéresser à la compression et décompression de texte !

Dictionnaires

Le module `Hashtbl` fournit des dictionnaires implémentés par tables de hachage.
Voici les fonctions qui peuvent vous intéresser :

- `Hashtbl.create n` renvoie un dictionnaire vide, implémenté par table de hachage. La table à `n` cases. Il faut choisir une valeur de `n` qui est une estimation du nombre de clefs que l'on va hacher [1].
- `Hashtbl.mem t k` renvoie `true` si `k` est une clef associée à une valeur dans `t`, `false` sinon.
- `Hashtbl.find t k` renvoie la valeur associée à la clef `k` dans le dictionnaire `t`. Si `k` n'est associée à aucune valeur, lève l'exception `Not_found` .
- `Hashtbl.find_opt t k` fait comme `find`, mais renvoie `Some v` avec `v` la valeur, et `None` si la clef n'est pas associée.
- `Hashtbl.add t k v` modifie `t` pour y ajouter l'association $k \mapsto v$. Si `k` était déjà associée, cette nouvelle association masque la précéente (mais ne l'écrase pas).
- `Hashtbl.replace t k v` fait comme `add` mais écrase l'éventuelle association précédente au lieu de la masquer.
- `Hashtbl.remove t k` supprime l'association (actuelle) de `k` dans `t`. Si `k` n'était pas associée, ne fait rien.
- `Hashtbl.length t` est le nombre d'associations de `t`. Si des associations en masquent d'autres, elles sont *toutes* comptées (y compris les masquées).

Vous pouvez aussi utiliser `iter` et `fold`. Pour plus d'infos : `https://v2.ocaml.org/api/Hashtbl.html`

A.2 Compression par code binaire

Rappelons que pour compresser un message, par exemple "hello world" [2], on va associer à chaque lettre du message son code binaire associé. Dans cet exemple, on a :

$$f : \begin{cases} h \mapsto 1110000 \\ e \mapsto ??? \\ l \mapsto 11101 \\ o \mapsto 11110 \\ \texttt{""} \mapsto ??? \\ w \mapsto 0111110111 \\ r \mapsto 0001 \\ d \mapsto 11010 \end{cases}$$

La première étape consiste donc à associer à chaque lettre son code binaire associé. On pourrait le retrouver directement dans l'arbre à chaque étape, mais ce serait très long. On va créer un dictionnaire dont les clés sont les lettres possibles, auquelles on associe comme valeur leur code binaire.

6. Écrire une fonction
   `arbre_to_dico : arbre_code -> (char, string) Hashtbl.t` telle que `arbre_to_dico arbre` renvoie le dictionnaire des associations lettre $\mapsto$ code binaire obtenues par le code binaire dont on a donné l'arbre. On le fera en une seule lecture de l'arbre.

1. Si vous n'en avez aucune idée ; choisissez votre entier strictement positif favori. La table est redimensionnée dynamiquement quand il y en a besoin.
2. Oui je sais, ce message n'est pas en Français. Compressons-le tout de même.

7. Écrire une fonction `compresse_message : string -> arbre_code -> string` qui, étant donné un message `m` à compresser et l'arbre du code binaire utilisé pour le compresser, renvoie le message obtenu après compression. Le message renvoyé ne devrait donc contenir que des caractères '0' et '1'.
8. Vérifiez que la compression du message "hello world" vous donne bien "11100000011110111011111010011111011111100001111011010".

A.3 Décompression

Pour la décompression, pas besoin de dictionnaire. On va lire le mot compressé en suivant l'arbre du code en parallèle. À chaque fois qu'on voit un "0", on descend dans le fils gauche. À chaque fois qu'on voit un "1", on descend dans le fils droit. Lorsqu'on a atteint une feuille, on lit le caractère qu'elle contient et on repart de la racine de l'arbre pour décoder le caractère suivant.

9. Écrire une fonction `decompresse_message : string -> arbre_code -> string` qui décode un message compressé et renvoie le message originel.
   Une complexité quadratique est acceptée ici, mais vous pouvez atteindre une complexité linéaire avec l'astuce de concaténation donnée précédemment.
10. `francais.ml` contient des messages encodés. Décodez-les !

B Algorithme de Huffman

Venons-en à l'algorithme de ce TP qui est au coeur du programme : l'algorithme de Huffman. Il ne se réduit pas à la simple compression/décompression avec un arbre prédéfini. Nous l'avons vu en cours : l'algorithme de Huffman consiste à créer un arbre de code binaire **optimal** pour le message que vous tentez de faire passer.

Cette section est plus ouverte : il y a plusieurs façons de faire. Ne restez pas bloqués plus de 10min à vous demander *quoi* faire : appelez-moi ![3]

Commençons par recréer l'arbre à partir de fréquences dans la langue française. Nous allons dans un premier temps nous appuyer sur les lettres et de fréquences fournies dans `francais.ml` pour tester les fonctions.

11. Écrire une fonction `codage_huffman : (char*float) list -> arbre_code` qui prend en argument une liste de couples (lettre, fréquence d'apparition) et qui construit l'arbre binaire de code obtenu par l'algorithme de Huffman.
    Indication : Commencer par transformer chaque lettre en arbre constitué d'une seule feuille, accompagné de sa fréquence. Utiliser les files de priorités min fournies.
12. Écrire une fonction `trouve_frequences : string -> (char * float) list` qui, étant donné un texte en argument, construit la liste des caractères qui y apparaissent avec leurs fréquences d'apparition.
    Indication : vous pouvez supposer que tous les caractères utilisés seront des caractères ASCII standards (au nombre de 128), et calculer les fréquences dans un tableau de 128 cases. On passe d'un caractère à un indice (entier compris entre 0 et 127) et inversement via les fonctions `int_of_char` *et* `char_of_int` *. Enfin, il restera à construire la liste de couples à partir de ce tableau.*

3. Je fais un très bon canard en plastique si je puis me permettre.