Devoir maison n°3 - À rendre le 18 Avril 2022
Compression de texte - partie 2 : Algorithme de Huffman

Notions abordées

- Algorithme de Huffman : principe, validité, complexité
- Implémentation de tas binaire par tableau
- Implémentation et expérimentation de la compression par codage de Huffman

Le code de ce projet peut être réalisé seul, en binôme ou en trinôme, mais les questions à rédiger sur papier (✍) doivent l'être individuellement. Les questions ✍ 🪜 peuvent être omises du rendu papier, qui se fera en classe. Le rendu de code se fera quant à lui sur cahier de prépa.

Toutes les fonctions doivent être commentées et **testées**. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.
En OCaml, les commentaires s'écrivent comme suit `(*Commentaires*)`.

Algorithme de Huffman

On utilisera ici les mêmes notations que dans la partie 1 de ce devoir. Pour rappel $\mathcal{C}$ désigne un ensemble fini de caractères, et Σ désigne l'alphabet $\{0,1\}$. On suppose que le cardinal N de $\mathcal{C}$ est supérieur ou égal à 2. On cherche un codage à taille variable de $\mathcal{C}$ sur Σ, qui soit non ambigu et de score minimal pour une distribution f connue des éléments de $\mathcal{C}$, c'est-à-dire une fonction φ de $\mathcal{C}$ dans Σ^* telle que $\bar{\varphi}$, le prolongement de φ aux mots de $\mathcal{C}^*$, soit injective, qui minimise $\sum_{c\in\mathcal{C}} f(c)|\varphi(c)|$.

On a vu qu'un tel codage correspond à un arbre binaire dont les feuilles sont étiquetées par $\mathcal{C}$, plus précisément où ces feuilles sont en bijection avec $\mathcal{C}$, arbre qu'on appelle arbre de décodage (car il est utilisé pour décoder ou décompresser un texte de Σ^* et non pour encoder un texte de $\mathcal{C}^*$).

L'**algorithme de Huffman**, décrit ci-dessous en pseudo-code, produit un arbre de décodage optimal.

Dans le premier exercice, on s'intéresse à la terminaison, à la correction et à la complexité de cet algorithme. Dans le deuxième exercice, on réalise une implémentation de tas binaire afin d'obtenir une implémentation efficace de l'algorithme de Huffman. Dans le dernier exercice, on revient au problème de compression de texte, et on compare une compression réalisée avec un codage quelconque avec une compression réalisée avec le codage de Huffman.

Algorithme de Huffman

Entrées : $\mathcal{C}$ un ensemble de cardinal $N \geqslant 2$
f une distribution des éléments de $\mathcal{C}$

Sortie : un arbre à N feuilles étiquetées par les N éléments de $\mathcal{C}$ de manière à minimiser leur profondeur moyenne pour la distribution f

Construire T un tas vide (d'arbre binaires non vides aux feuilles étiquetées)

Pour chaque élément $c \in \mathcal{C}$:
  Construire un arbre réduit à une feuille étiquetée par c
  Insérer cet arbre dans T avec la priorité $f(c)$

Tant que taille$(T) > 1$:
  $(e_1, p_1) \leftarrow \min(T)$
  supprime_min(T)
  $(e_2, p_2) \leftarrow \min(T)$
  supprime_min(T)
  Construire a l'arbre ayant pour fils gauche e_1 et fils droit e_2
  Insérer dans T l'arbre a avec la priorité p_1+p_2

Retourner l'unique arbre contenu dans T

Exercice 1 Comprendre l'algorithme

Question 1 ✍ 🪜
Exécuter l'algorithme de Huffman à la main pour l'ensemble $\mathcal{C} = \{\texttt{A,B,C,D,E,F}\}$ muni de la distribution f définie par le tableau ci-dessous.

c	A	B	C	D	R	!
$f(c)$	0.3125	0.125	0.625	0.625	0.125	0.3125

Question 2 ✍
Justifier que la boucle **tant que** de cet algorithme termine, et préciser combien de tours de boucles sont effectués. *La réponse à cette question pourra d'ailleurs permettre de remplacer la boucle tant que par une boucle pour lors de l'implémentation de cet algorithme.*

Question 3 ✍
En fonction des complexité des opérations d'insertion, de suppression du minimum et d'extraction du minimum sur le tas T, donner la complexité pire cas de l'algorithme de Huffman au moyen d'un O. En déduire la complexité de cet algorithme pour l'implémentation de tas qui vous paraît la plus adaptée.

Question 4 ✍
Quel invariant de boucle permet de justifier que l'arbre retourné par l'algorithme couvre bien $\mathcal{C}$, c'est-à-dire que chacun des caractères de $\mathcal{C}$ apparaît bien en feuille de l'arbre calculé.

Question 5 ✍
Justifier que l'algorithme de Huffman permet de calculer un codage non ambigu.

Question 6 ✍
Que reste-t-il à démontrer pour conclure que l'algorithme de Huffman est correct ?

Exercice 2 Implémenter l'algorithme

Dans un premier temps, on se concentre sur l'implémentation de l'algorithme de Huffman sans rentrer dans le détail de l'implémentation de tas utilisée. Pour cela, une implémentation de tas en Ocaml est fournie dans le fichier `tas_liste.ml`. Cette implémentation pourrait être qualifiée de naïve dans le sens où elle est simple, et où aucun effort n'a été fait pour réduire la complexité des opérations d'extraction et de suppression du minimum, toutes deux en $\Theta(n)$ pour n le nombre d'éléments du tas. Outre les fonctions listées ci-dessous, ce fichier contient la définition du type `'a tas_min` pour les tas d'éléments de type `'a` munis de priorités de type `float`.

```
- cree_tas_min_vide () : 'a tas_min
- est_vide (t:'a tas_min) : bool
- elem_min (t:'a tas_min) : 'a
- insere_tas (t:'a tas_min) (elem_vlr:'a*float) : unit
- supprime_min (t:'a tas_min) : unit
```

Pour modéliser les codages, les arbres de codage et de décodage, et les distributions on utilisera les types définis dans la première partie du devoir, rappelés ci-dessous

```
- type codage = bool list
- type arbre_d =
  | Feuille of char
  | Ndd of arbre_d * arbre_d
- type distribution = (char*float) list
```

Question 1 🖳
Dans un fichier nommé `huffman_liste.ml` définir une fonction nommée `fusionne` de signature `(af1:arbre_d*float)(af2:arbre_d*float):arbre_d*float` qui prend en entrée deux arbres de décodage a_1 et a_2 munis de leur priorité p_1 et p_2(de type `float`), et qui calcule l'arbre de fils gauche a_1 et de fils droit a_2, muni de la priorité p_1+p_2.

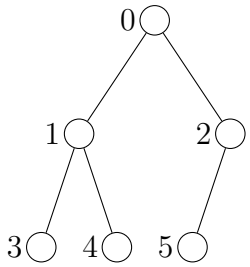
Question 2 🖳
Définir dans le même fichier une fonction nommée `huffman_liste` qui prend en entrée une distribution d et qui calcule un arbre de décodage optimal pour d selon l'algorithme de Huffman.

Question 3 ✍
Sachant que l'implémentation de tas donnée réalise l'insertion en $\Theta(1)$, mais la suppression et l'extraction du minimum en $\Theta(n)$, quelle est la complexité de la fonction `codage_huffman` ?

Question 4 ✍
Si on avait une implémentation de tas qui réalise l'extraction du minimum en $\Theta(1)$, l'insertion et la suppression en $\Theta(\log(n))$, quelle serait la complexité de l'algorithme de Huffman ?

Dans la suite de cet exercice on développe une implémentation de tas qui réalise les complexités mentionnées à la question précédente en vue d'obtenir une implémentation de l'algorithme de Huffman plus efficace.

Une implémentation concrète du tas binaire vu en cours consiste à stocker les éléments du tas, ainsi que leur priorité, dans un tableau selon leur position dans l'arbre. Plus précisément, on numérote les nœuds de l'arbre à partir de 0 pour la racine, puis de gauche à droite étage par étage (un étage étant un ensemble maximal de nœuds ayant la même profondeur). L'élément du nœud de numéro k et sa priorité sont alors stockés dans la case d'indice k du tableau. *La figure ci-contre présente la numérotation des nœuds d'un arbre parfait à 6 nœuds.*



Question 5 ✍ 🪜
Dans un arbre parfait à n nœuds, quel est le numéro du dernier nœud ? À quelle condition sur i le nœud de numéro i a 0, 1 ou 2 enfants, et quels sont alors les numéros de ses enfants.

Question 6 🪜
Dans un arbre parfait à n nœuds, préciser à quelle condition sur i le nœud de numéro i a un parent et quel est alors son numéro.

On se restreint ici à implémenter un tas min dont les éléments sont munis de priorités de type `float`. Le type des éléments eux-mêmes n'est pas fixé. De plus on admet qu'on connaît initialement un majorant du nombre d'éléments simultanément dans le tas, ce qui permet de fixer à la création du tas la taille allouée au tableau. Enfin, on supposera que la priorité d'un élément du tas n'est jamais modifiée, c'est-à-dire qu'on ne peut pas "mettre à jour" la priorité d'un élément. Excepté la dernière, toutes les questions qui suivent sont à coder dans un fichier nommé `tas_min.ml`.

Question 7 ✍
Ces deux dernières hypothèses paraissent-elle justifiées dans le cadre de l'algorithme de Huffman ?

Puisqu'on fixe initialement la taille du tableau, certaines cases de ce tableau représentent des cases vides, c'est-à-dire que la valeur stockée dans une telle case en machine n'est pas pertinente. On remarque que les cases vides sont toutes placées consécutivement en fin de tableau. Afin de différencier une case pleine d'une case vide, on enregistrera donc dans la structure de tas le nombre d'éléments qu'il contient.
De plus en OCaml, à l'initialisation d'un tableau de type `t`, toutes les cases doivent avoir une valeur de type `t`. Afin de ne pas avoir à inventer un élément factice de type `t` pour remplir ces cases vides, on utilisera un type `option`. La valeur `None` désignera une case vide, tandis qu'une valeur `Some(e,p)` encapsulera un élément du tas, `e`, et sa priorité `p`. Cependant, même si on sait que la case d'indice `i` de `tab` est non vide, on évitera de récupérer son contenu par `let Some(e,p) = tab.(i)`, car cela déclenche un avertissement indiquant qu'on ne traite pas le cas `None`. À la place, on écrira plutôt `let (e,p) = get(tab.(i))` profitant de la fonction `Option.get` définie comme suit dans la librairie standard. `let get = function Some v -> v | None -> invalid_arg "option is None"`

Question 8 💻
Définir le type paramétré `'a tas_min`.

Question 9 💻
Définir une fonction `cree_tas_min_vide` qui crée un tas vide à partir du nombre maximal d'élément que celui-ci pourra contenir.

Question 10 💻
Définir une fonction `est_tas_vide` (resp. `est_tas_plein`) qui teste si un tas est vide (resp. plein).

Question 11 💻
Définir une fonction `ajoute_feuille` qui prend en argument un élément et sa priorité sous forme de couple ainsi qu'un tas, et qui ajoute en feuille cet élément dans le tas. *Cette fonction est une fonction intermédiaire, elle ne fait qu'ajouter l'élément en dernière feuille, sans considérer la priorité de l'élément.*

Question 12 💻
Définir une fonction `elem_prio_min` qui calcule le couple élément/priorité de priorité minimum d'un tas.

Question 13 💻
Définir une fonction `est_valide` qui prend en argument un tas, et qui teste si celui-ci est bien un tas min, c'est-à-dire que la priorité de chaque nœud est supérieure ou égale à celles de ses enfants s'il en a. *Cette fonction, qui a vocation à n'être utilisée que lors des tests en phase de développement peut être codée de manière récursive non terminale.*

Question 14 💻
Définir une fonction `remonte_feuille` qui prend en argument un tas, et qui remonte la dernière feuille par échange successifs tant que sa priorité l'exige. Plus précisément, le tas pris en argument est valide si l'on omet sa dernière feuille, qui est donc le seul élément potentiellement mal placé.

Question 15 💻
Définir une fonction `insere_tas` qui prend en argument un élément et sa priorité sous forme de couple ainsi qu'un tas, et qui ajoute cet élément dans le tas à une position qui convient selon sa priorité.

Question 16 💻
Définir une fonction `descend_racine` qui prend en argument un tas, dont la racine est éventuellement mal placée pour sa priorité, mais dont les sous-arbres stricts sont bien tous des tas valides, et qui descend cet élément par échanges successifs jusqu'à obtenir un tas valide.

Question 17 💻
Définir une fonction `supprime_min` qui prend en argument un tas et qui en supprime l'élément de priorité minimum. *Cette fonction doit donc réaliser les déplacements nécessaires pour que le tas soit valide en sortie.*

Question 18 💻
Définir dans un fichier nommé `huffman.ml` une fonction `huffman` qui prend en entrée une distribution `d` et qui calcule un arbre de décodage optimal pour `d` selon l'algorithme de Huffman en utilisant l'implémentation de tas binaire par tableau.

Exercice 3 Utiliser l'algorithme de Huffman

Dans cet exercice on revient au problème de compression de texte qui était notre point de départ dans la première partie du devoir. On mesurera de manière empirique les taux de compressions pour différents codages, on les comparera entre eux, et aux valeurs théoriques attendues.

Pour réaliser des compressions réalistes, et pas seulement sur des chaînes de caractères relativement courtes car saisies manuellement, plusieurs fichiers sont fournis sur cahier de prépa :

- le fichier texte `bonjour_tristesse_debut.txt`[1] qui est relativement court ;
- le fichier texte `les_miserables_partie_1_livre_1.txt`[2] qui est quant à lui assez long ;
- le fichier `outils.ml` qui contient le code de plusieurs fonctions permettant l'interface entre les objets manipulés en Ocaml et les données enregistrées dans un fichier.

Dans un premier temps, on utilisera `string_of_file` de signature `(filename: string): string` pour importer le contenu d'un fichier texte sous la forme d'un `string` en Ocaml, et sa réciproque `file_of_string` de signature `(filename: string) (s: string): unit`.

Question 1
Copier le fichier `bonjour_tristesse_debut.txt` dans un fichier `sagan.txt` en faisant un import du texte sous Ocaml puis un export. On peut aussi recopier `les_miserables_partie_1_livre_1.txt` dans un fichier nommé `hugo.txt` pour s'assurer que l'on peut gérer des chaînes aussi longues.

Question 2
Dans un fichier nommé `compression.ml` définir une fonction `calcule_distribution` qui prend en entrée une chaîne de caractères et retourne la distribution correspondante. Par exemple pour la chaîne `"ABRACADABRA!!!!!"` on doit retrouver la distribution donnée en exemple à la question 1 de l'exercice 1. *Sachant que les caractères sont encodés sur un octet, on remarque que la fonction* `int_of_char` *établit une bijection entre l'ensemble des caractères et l'intervalle* $[0..255]$*, de réciproque* `char_of_int`*. On peut donc d'abord compter les occurrences de chacun des 256 caractères existants en parcourant la chaîne, puis former la liste des couples (caractère,fréquence) pour les caractères présents.*

Question 3
Calculer `d_sagan` la distribution des caractères du fichier `sagan.txt`. *Cette distribution doit compter 57 caractères.*

Question 4
Calculer un encodage optimal pour la distribution `d_sagan` avec l'algorithme de Huffman.
À l'aide de fonctions développées dans la partie 1 du devoir, calculer le score de cet encodage. En déduire le nombre de bits nécessaires pour encoder `sagan.txt` à partir de son nombre de caractères[3].

[1] *Bonjour tristesse* de Françoise Saqan, disponible aux éditions Julliard

[2] Ce fichier est extrait du texte intégral du roman *Les misérables* de Victor Hugo disponible sur wikisource à l'adresse suivante. `https://fr.wikisource.org/wiki/Les_Misérables`

[3] Ce nombre de caractères s'obtient en appliquant la fonction `String.length` à la chaîne résultant de l'import de ce fichier. Il est déconseillé de compter les caractères à la main. Au risque évident de faire une erreur de comptage s'ajoute l'erreur légitime qui consiste à compter 1 caractère pour `è` alors qu'il est en réalité traité comme deux caractères spéciaux qui, lorsqu'ils se suivent, sont affichés sous la forme d'un seul : `è`. Nos fonctions traitent ces deux caractères spéciaux séparément, et tout fonctionne bien, mais à l'œil nu on voit 1 caractère là où il y en a 2.

Question 5 🖳 🪜

Comme expliqué dans le corrigé de la partie 1, on peut transformer un arbre de décodage en une table d'encodage qui permet de retrouver en temps constant le codage d'un caractère et ainsi de réaliser plus efficacement le codage, c'est-à-dire ici la compression. La fonction `tab_e_from_arbre_d` fournie dans le fichier `outils.ml` effectue cette transformation. Tester cette fonction pour calculer la table d'encodage associée au codage fourni par l'algorithme de Huffman pour `sagan.txt`.

Question 6 🖳 🪜

Modifier la fonction `code` précédemment développée pour qu'elle prenne en argument une table d'encodage (plutôt qu'une `structure_e` ou un `arbre_d`) et qu'elle appelle la fonction `cod_from_char`, fournie dans le fichier `outils.ml`, qui calcule le codage d'un caractère à partir d'une table d'encodage. Tester cette fonction `code`.

On souhaite maintenant comparer la taille de la séquence de bits du texte compressé à celle du texte initial. À ce stade on a d'une part le texte initial, enregistré dans un fichier sous la forme d'une séquence d'octets codant chacun pour un caractère, et d'autre part un objet de type `codage`, c'est-à-dire une liste de booléens dans notre interpréteur Ocaml. On obtient facilement une taille en ko (kilo-octets) du fichier, soit dans le navigateur de fichiers, soit en tapant `ls -l` dans un terminal ouvert dans le dossier concerné (dans la colonne avant le mois se trouve la taille du fichier en octets). On obtient aussi facilement la taille en nombre de booléens de la liste grâce à la fonction `List.length`. Le problème est que ces deux tailles ne sont pas comparables. De plus on aimerait pouvoir enregistrer le texte compressé, pour le manipuler ensuite comme un fichier quelconque (copier/coller, dépôt en ligne, mise en pièce jointe...).

Une première idée pourrait être d'exporter notre liste de booléens dans un fichier texte constitué de `0` et de `1`. Ce serait facile, et lisible, mais néanmoins une très mauvaise idée : en effet, chaque booléen serait alors codé en mémoire sur un octet, soit 8 bits, comme n'importe quel caractère, alors qu'un octet peut encoder jusqu'à 8 booléens. Pour éviter cet écueil on utilisera les fonctions `file_with_mod_of_bit_sequence` et `bit_sequence_of_file_with_mod`, fournies dans le fichier `outils.ml`, et réalisant respectivement l'écriture et la lecture d'un fichier enregistrant un codage[4].

Question 7 🖳 ✍

Exporter le codage du texte `sagan.txt` dans un fichier nommé `sagan.comp`. Comparer la taille de ce fichier au fichier initial `sagan.txt`. La compression a-t-elle été à la hauteur de ce qui était attendu ?

Question 8 🖳

Définir une fonction `compresse (inputfile:string)(outputfile:string): arbre_d` qui prend en argument le nom d'un fichier texte qui existe, et le nom d'un fichier qui n'existe pas (ou qu'on est prêt à écraser) et qui compresse dans le fichier `outputfile` le texte contenu dans le fichier `inputfile`, grâce au codage de Huffman. De plus cette fonction retourne l'arbre de décodage utilisé, en vue d'une décompression ultérieure. *Il s'agit essentiellement de mettre bout à bout les étapes détaillées jusqu'ici.*

Question 9 🖳 ✍

Quel est le taux de compression observé pour la compression `hugo.txt` ? Correspond-il au taux attendu d'après le score du codage de Huffman pour la distribution des caractères dans `hugo.txt` ?

[4]Grâce à ces fonctions on enregistre notre liste de booléens par tranche de 8 booléens, chacune correspondant à un octet, sauf éventuellement la dernière tranche si elle contient moins de 8 bits (penser à une liste de $35 = 8+8+8+8+3$ booléens par exemple). À l'écriture ce n'est pas trop gênant, mais si on veut pouvoir retrouver exactement notre liste initiale à partir du fichier, il faut enregistrer combien de bits du dernier octet sont significatifs (3 dans l'exemple). Cette information étant un entier entre 1 et 8, elle peut tout à fait être inscrite en début de fichier, dans le premier octet. Ainsi l'export d'une liste de 35 booléens résulte en un fichier de 6 octets dont le premier est `00000101` pour indiquer que seuls 3 bits du 6ème octet sont significatifs, et que les autres doivent être ignorés à la lecture.

Afin de s'assurer que la procédure de compression n'entraîne aucune perte d'information, on vérifie qu'on peut décompresser le fichier compressé et retrouver le texte initial.

Question 10 🖳

Définir une fonction `decompresse (inputfile:string)(outputfile:string)(ad:arbre_d):unit` qui prend en argument le nom d'un fichier compressé qui existe, et le nom d'un fichier qui n'existe pas (ou qu'on est prêt à écraser) et qui décompresse selon `ad` le contenu du fichier `inputfile`, dans le fichier `outputfile`.

Question 11 🖳 ✍ ∗

Proposer un moyen de constater empiriquement que choisir l'encodage de Huffman, plutôt qu'un encodage à taille variable quelconque (*i.e.* pas particulièrement de score minimal), apporte une réelle plus-value en terme de compression.