Compression de Huffman

A rendre le jour de la soutenance (vendredi 16 janvier 2015)

Le projet est à réaliser en OCaml individuellement. Il sera accompagné d'un dossier contenant impérativement la description des choix faits, la description des types et des fonctions. Pour chaque fonction, on donnera impérativement l'interface complète (dans le code en commentaire et dans le rapport pour les fonctions présentées).

Même si le sujet est décomposé en questions, en général chaque question se résoud par l'écriture d'une ou plusieurs fonctions intermédiaires. Celles-ci doivent comporter une interface également.

Le dossier fournira également des cas de tests accompagnés des résultats attendus et retournés.

On s'intéresse à l'implantation d'un algorithme de compression de textes appelé codage de Huffman. On cherche à coder, par une liste de '0' et de '1' aussi courte que possible, un texte comportant n caractères différents. Un moyen efficace d'y parvenir est d'utiliser la méthode du codage Huffman qui prend en compte la fréquence de chaque caractère intervenant dans le texte afin d'associer les codes les plus cours aux caractères les plus fréquents. Ce codage utilise une correspondance entre les listes de booléens (appelés codes dans la suite) et les listes de caractères quelconques (appelé textes dans la suite). Cet algorithme se divise en deux fonctions distinctes : la décompression et la compression.

1 Outils

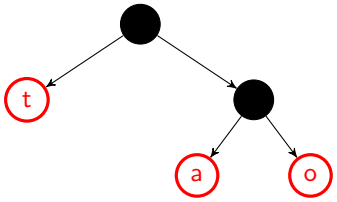
Cette partie est composée des diverses fonctions qui ne sont pas spécifiques au codage de Huffman mais qui seront utilisées dans la suite.

1. Définir un type `arbre` pour les arbre binaires comportant des étiquettes que sur les feuilles.
2. Écrire une fonction `nb_noeuds` qui calcule le nombre de nuds internes dans un arbre.
3. Écrire une fonction `nb_feuilles` qui calcule le nombre de feuilles d'un arbre.
4. Écrire une fonction `trouve` qui prend en arguments une liste de couples et un élément x et cherche dans la liste un couple dont le premier élément est x et dans ce cas renvoie le second élément ou bien renvoie une erreur s'il n'y a aucun couple avec x comme premier élément.
5. Écrire une fonction `insertion` qui permet d'insérer un élément dans une liste triée (on prendra la relation d'ordre en argument).

2 Décompression

On appelle la traduction d'un code v en un texte w la décompression de v en w. La méthode de Huffman se sert d'un arbre binaire aux feuilles duquel se trouvent les caractères à coder. La décompression applique l'algorithme suivant : elle part de la racine de l'arbre et suit le chemin indiqué par le booléen (`false` pour gauche et `true` pour droit). Dès qu'on arrive à une feuille, on ajoute au résultat le caractère associé à cette feuille et on recommence le décodage à la racine de l'arbre. Quand le code est parcouru en entier, on a reconstruit le texte.

Par exemple, considérons l'arbre :



La décompression du code `[false;true;false;false;true;true]` donne le texte `['t';'a';'t';'o']`.

1. Écrire une fonction `cherche_car` qui prend en arguments un arbre et un code et renvoie le premier caractère encodé par ce code ainsi que le reste du code.
2. Écrire une fonction `decode` qui prend en argument un arbre et un code et renvoie la liste des caractères encodés par ce code.

3 Compression

L'algorithme de compression se décompose en deux étapes : d'abord la construction d'un arbre, puis l'encodage du texte au moyen de l'arbre.

3.1 Construction de l'arbre

L'idée générale de la construction de l'arbre de Huffman est la suivante :
— on part d'une liste de caractères accompagnés de leur fréquence d'apparition
— on construit une liste de paires constituées d'un arbre et d'un poids (au départ ces arbres sont réduits à une feuille)
— on extrait de cette liste deux arbres de poids minimaux et on les fusionne en un nouvel arbre dont les sous-arbres sont les deux arbres choisis et le poids la somme de leur poids
— on répète l'étape précédente jusqu'à ce qu'il ne reste plus qu'un seul arbre.

1. Écrire une fonction `frequences` qui prend en argument un texte et qui retourne une liste de couples (n, c) tel que l'entier n est le poids du caractère c dans le texte. Cette liste doit être triée dans le sens des poids croissants. On utilisera la fonction `List.sort` de la bibliothèque standard. Par exemple, appliquée au texte `['e';'s';'s';'a';'i']` la fonction `frequences` renvoie `[('e',1);('a',1);('i',1);('s',2)]`.

A partir de cette liste de fréquences, l'algorithme de compression construit un arbre binaire dont chaque feuille contient un caractère du texte. Cet arbre est tel que chaque caractère du texte est contenu dans une et une seule feuille. La construction de l'arbre est réalisée de la manière suivante. L'algorithme travaille sur une liste de couples (entier, arbre). On appelle cet entier le poids de l'arbre. Étant donné la liste de poids des caractères, on commence avec la liste `[(p1,a1); ... ; (pn,an)]` où a_i est l'arbre qui contient seulement la feuille c_i. Tant que la liste contient plus d'un élément, on sélectionne les deux couples de plus faible poids et on les enlève de la liste. Il est important de remarquer que les deux éléments de plus faible poids sont toujours les deux éléments en tête de la liste. On construit un arbre dont les deux fils sont les deux arbres des deux couples que l'on vient d'enlever. On calcule un nouveau poids pour cet arbre qui est la somme des poids de ses deux sous-arbres. On rajoute le couple (arbre-fusionné, poids-calculé) à la liste dont on avait enlevé les deux couples de plus faible poids. On s'assure durant cette insertion que a liste reste triée par ordre de fréquence croissant. Dès que la liste contient un seul couple (a, p) l'algorithme renvoie l'arbre a.

1. Écrire une fonction `initialiste_liste_arbre` qui prend en argument une liste de couples (fréquence,caractère) et renvoie une liste de couples formés de la fréquence f et de l'arbre contenant une seule feuille c.
2. Écrire une fonction `creer_arbre` qui crée l'arbre de Huffman.

3.2 Encodage

1. Pour coder un texte, l'arbre lui-même est peu pratique à utiliser pour trouver le code d'un caractère. Donc, on calcule à partir d'un arbre une liste `[(c1,m1); ... ; (cn,mn)]` où les

c_i sont tous les caractères contenus dans l'arbre et m_i est une liste représentant le chemin d'accès pour le caractère c_i. Par exemple, pour l'arbre donné au-dessus, on obtient la liste `('t',[false]),('a',[true;false]);('o',[true;true])]` (l'ordre dans la liste n'importe pas). Écrire une fonction `creer_traduction` qui prend un arbre en argument et qui renvoie la liste des couples (caractère, chemin d'accès) comme décrit au-dessus.

2. Écrire une fonction `encode_liste` qui prend une structure de données tel que décrits dans la question précédente et une liste de caractères et renvoie une liste de booléens représentant le codage des caractères.
3. Finalement, écrire une fonction qui prend en argument un texte et retourne le couple (code,arbre).
4. Tester.

4 Extension : gestion des entrées/sorties

Mettre en place un mécanisme d'entrées-sorties pour lire, compresser, écrire, décompresser des fichiers texte.

1. Lire un fichier depuis le disque
2. Construire l'arbre et compresser les données (ne pas oublier de stocker l'arbre pour pouvoir décoder ensuite)
3. Lire un fichier compressé sur le disque (et l'arbre associé)
4. Le décompresser

Bien penser à vérifier que les fichiers compressés sont effectivement plus compacts que les fichiers initiaux (sinon réfléchir aux raisons de cette bizarrerie...). Il est notamment déconseillé de coder un seul bit dans une variable de type `int`.

Les plus ambitieux pourront s'attaquer à la compression/décompression de fichiers binaires (images, code compilé, etc.) et analyser les performances obtenues (exécution correcte, temps d'exécution, taux de compression, comparaison avec les outils de compression usuels, etc.)