TP 1: Arbres de Huffman

Remarques préliminaires

Structure du projet

Ce TP s'étale sur deux ou trois séances. Il s'agit d'un projet conséquent dont le but est d'arriver à une compression de données à l'aide du codage de Huffman vu en cours. Les principales étapes sont les suivantes:

- Calcul des fréquences d'un texte (donné en forme d'une chaîne de caractères), voir § 2, qui permet de construire un arbre de Huffman (§ 3).
- Utilisation d'un arbre de Huffman pour la compression d'un texte.
- L'opération inverse qui permet de restituer le texte original (§ 4).

Pour vous familiariser avec la structure de données d'arbres, il y a une section d'introduction à la manipulation des arbres (§ 1). Vous pouvez en principe aborder chacune des sections indépendamment, mais pour faire le tour complet (texte original → texte codé → texte original), il faut observer quelques dépendances entre les parties.

Avant de commencer, récupérez l'archive `huffman.tar` de Moodle. Les fichiers que vous y trouvez seront utilisés au fur et à mesure. Une partie du code est déjà donnée, le reste est à compléter par vous. Pour ne pas se perdre dans les procédures (250-300 lignes de code Python), il est important de structurer le code en paquetages. Le mécanisme des paquetages est décrit en annexe B.

Récupérer et extraire l'archive

Nous supposons dans la suite que vous travaillez sous **Linux**. Pour commencer à travailler, procédez comme suit:

- Ouvrez une console[1]
- Faites les préparatifs nécessaires pour ne pas vous perdre dans un fouillis de fichiers: créez un répertoire pour le TP de "Théorie de l'information" (par exemple avec `mkdir TheorieInf`) et déplacez l'archive `huffman.tar` dans ce répertoire.
- Extrayez l'archive, avec `tar -xf huffman.tar`

Vous avez ensuite la possibilité de travailler avec l'interpréteur Python localement sur votre machine, ou en utilisant `repl.it` à distance. Cette dernière option n'est pas la plus adéquate (voir annexe C). Pour cela, nous décrivons dans la suite l'utilisation de Python localement sur votre machine.

[1] tapez `Ctrl-C` pour faire disparaître le message vous demandant de saisir un mot de passe.

Travail avec l'interpréteur de Python

Avant-propos: Le "service informatique" de cette université n'a pas la volonté ou est incapable d'installer le logiciel nécessaire pour un déroulement correct de ce TP, et de corriger les inepties les plus flagrantes qu'on lui signale depuis des années. Nous vous décrirons plus tard dans le texte comment contourner certains problèmes. Merci d'avance de votre patience.

Pour travailler localement avec l'interpréteur de Python:

- Descendez dans le répertoire contenant les fichiers: `cd Huffman`
- Lancez l'interpréteur: `python`. La version devrait être > 3 (message du style `Python 3.4` ou `Python 3.6`). Vous êtes maintenant dans le mode Python (avec une invite >>>). Si, en plus, vous avez besoin de manipuler des fichiers, ouvrez un éditeur classique.

1 Arbres [60 min]

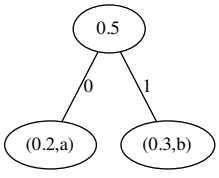
Les exercices de cette section ont pour but de vous familiariser avec la construction d'arbres et de différentes formes de parcours d'arbres. Le code de cette section ne sera pas utilisé pour les autres exercices.

Ouvrez le fichier `arbres.py` qui inclut différentes autres définitions. N'essayez pas de comprendre le code de `htrees.py`; tout ce qui importe ici est la signature des deux constructeurs qui permettent de construire des arbres (de type `Htree`):

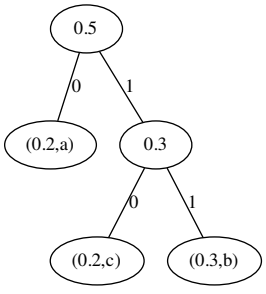
- `Leaf (val, code)` qui prend une valeur `val` (un flottant) et un `code` (un caractère), et qui construit une feuille.
- `Node (val, low, high)` qui prend une valeur `val` (un flottant) et deux arbres `low` et `high` (des `Htree`), et qui construit un noeud intérieur.

L'annexe A discute deux notations différentes: l'écriture vue en cours et l'écriture de fonctions en Python. Pour commencer à travailler, il suffit de coller le contenu du fichier `arbres.py` dans l'interpréteur de Pyton. Définissez les autres fonctions de cette section dans le fichier `arbres.py`.

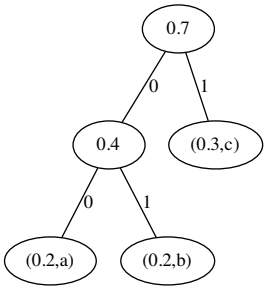
Exercice 1 Écrivez les expressions de constructeur qui correspondent aux arbres des figures 1. Le premier arbre est déjà donné dans le fichier.



(a) Arbre 1



(b) Arbre 2



(c) Arbre 3

Figure 1: Arbres de Huffman

Exercice 2 Testez les expressions de l'exercice 1, par un appel à la fonction `display_htree`, par exemple `display_htree(a1)`.

La librairie pour la visualisation des arbres n'a pas été installée correctement sur les machines. Un appel comme `display_htree(a1)` produit un code utilisé par un autre programme[2] qui effectue l'affichage. Vous pouvez aussi utiliser ce programme dans un navigateur:

- Allez sur http://graphviz.it/
- Collez tout le code `graph { ... }` produit par `display_htree` dans cet interpréteur (éventuellement, il faut d'abord supprimer le code qui s'y trouve).

Exercice 3 Écrivez les fonctions suivantes:

1. la fonction `profondeur` qui prend un arbre et calcule sa profondeur (le plus long chemin menant de la racine à une feuille). Ainsi, l'arbre 1 de la figure 1 a la profondeur 1 et les arbres 2 et 3 la profondeur 2.

   La fonction est une fonction récursive. Demandez-vous

   - quelle est la profondeur d'un arbre qui consiste uniquement d'une feuille
   - comment vous pouvez déterminer la profondeur d'un arbre si vous connaissez les profondeurs de ses sous-arbres.

2. la fonction `membre` qui prend un arbre et un caractère et vérifie si le caractère apparaît dans une feuille de l'arbre.

3. la procedure `afficher_arbre` qui prend un arbre et l'affiche de manière textuelle, par exemple (pour l'arbre 2):

   ```
   0.50000
       0.20000: a
       0.30000
           0.20000: c
           0.30000: b
   ```

   Vous verrez que l'arbre n'est pas le seul paramètre de la procédure, mais que vous avez aussi besoin d'un niveau d'indentation (un entier) qui est 0 pour la racine et incrémenté pour chaque sous-arbre.

Exercice 4 Écrivez les trois fonctions suivantes: `nb_feuilles` (compte le nombre de feuilles d'un arbre), `nb_nds_int` (compte le nombre de noeuds intérieurs ayant le constructeur `Node`), `nb_noeuds` (compte le nombre de tous les noeuds, `Leaf` et `Node`). Testez ces fonctions.

Étant donné un arbre `t`, quel doit être le rapport entre `nb_feuilles(t)`, `nb_nds_int(t)` et `nb_noeuds(t)`? Testez que ce rapport est bien satisfait pour vos fonctions.

Remarque: Bien entendu, avant de passer à l'exercice suivant, vous devez tester exhaustivement les fonctions que vous venez d'écrire.

2 Fréquences de caractères dans un texte [20 min]

Pour tous les exercices suivants, complétez le code du fichier `htree_construct.py`.

Une distribution de fréquences de caractères dans un texte mesure la proportion d'occurrences de chaque caractère dans le texte. Par exemple, dans la chaîne `abacd`, la fréquence de la lettre `a` est 0.4, des lettres `b`, `c` et `d` est 0.2 respectivement, et celle de toute autre lettre est 0.

[2] Graphviz, http://graphviz.org/

La distribution de fréquences sera réalisée par un dictionnaire[3] en Python. Nous rappelons qu'un dictionnaire associe des valeurs à des clés. Un dictionnaire vide s'écrit `{}`; pour un dictionnaire `d`, on accède à la valeur associée à une clé `k` avec `d[k]`, et on défini (ou met à jour) la valeur `v` pour la clé `k` avec `d[k] = v`. Vous pouvez tester si une valeur est associée à une clé `k` avec le test `k in d`. Les clés et valeurs peuvent appartenir à n'importe quel type scalaire.

Exercice 5 Supposez que vous avez un dictionnaire qui associe des entiers à des clés (dans la suite, ces clés seront des caractères). Écrivez la procédure `increment` qui incrémente le nombre associé à une clé, ou rajoute la clé au dictionnaire:

```
>>> d = { 'a': 3, 'b': 2 }
>>> d['a']
3
>>> increment(d, 'a')
>>> d
{'a': 4, 'b': 2}
>>> increment(d, 'c')
>>> d
{'a': 4, 'c': 1, 'b': 2}
>>> 'a' in d
True
>>> 'z' in d
False
```

Exercice 6 Écrivez la fonction `ch_distrib` qui calcule la distribution de fréquences d'une chaîne de caractères, par exemple:

```
>>> ch_distrib ("abacbadbdb")
{'a': 0.3, 'c': 0.1, 'b': 0.4, 'd': 0.2}
```

Astuce 1: Vous pouvez itérer sur tous les caractères d'une chaîne `str` avec `for c in str: ...`
Astuce 2: Vous pouvez itérer sur toutes les clés définies d'un dictionnaire `d` avec `for k in d: ...`

Exercice 7 Testez la fonction avec un texte plus large, par exemple des extraits de *Alice in Wonderland* (texte anglais) et *Les Misérables* (texte français, avec caractères accentués) qui se trouvent aussi dans l'archive.

Pour le faire, copiez le début du fichier `huffman.py` (y compris la définition de la fonction `read_file_latin1`) dans l'interpréteur Python. Vous pouvez ensuite récupérer le contenu du fichier nommé `f.txt` avec `read_file(f)`. Pour tester la fonction `ch_distrib`, vous pouvez donc faire:

```
 >>> ch_distrib(read_file_latin1("miserables"))
{'ê': 0.0030653950953678476, "'": 0.010728882833787467,
 't': 0.06079700272479564, 'M': 0.005619891008174387, ...}
```

Remarque: Les deux échantillons de texte sont codés en Latin-1 (ISO 8859-1), contrairement aux originaux sur www.gutenberg.org (en UTF-8). Les fonctions d'écriture / lecture de fichiers dans `huffman.py` acceptent uniquement ce codage. L'alphabet utilisé a donc des caractères avec des codes $0 \ldots 255$ (tous les caractères représentables par un octet).

[3] https://docs.python.org/3/tutorial/datastructures.html#dictionaries

3 Construction d'un arbre de Huffman [100 min]

3.1 Étapes de construction de l'arbre

Nous rappelons que l'algorithme de Huffman consiste à combiner des arbres contenus dans un ensemble d'arbres, jusqu'à obtenir un seul arbre. Nous allons représenter cet ensemble par une liste.

Exercice 8 Écrivez la fonction `init_treeset` qui prend une distribution de caractères comme définie dans la section 2 et construit un ensemble d'arbres initial (c'est à dire, une feuille par caractère de l'alphabet qui apparaît dans la distribution de caractères.

La fonction est facile à écrire si vous vous souvenez de l'itération sur les clés d'un dictionnaire (voir exercice 6).

Exercice 9 Une opération essentielle de la construction est la combinaison de deux arbres construits précédemment en un nouvel arbre où la probabilité cumulée est la somme des probabilités des sous-arbres (référez-vous aux transparents du cours pour le principe). Écrivez la fonction `combine_two` qui prend deux arbres et construit un nouvel arbre selon ce principe.

Exercice 10 La fonction `combine_all_trees` que vous écrivez dans cet exercice effectue l'essentiel de la construction. Il s'agit d'une fonction qui prend une liste d'arbres et qui renvoie un seul arbre, à savoir l'arbre de Huffman complet. (Initialement, la liste des arbres sera construite par la fonction `init_treeset` de l'exercice 8, mais ceci ne joue aucun rôle ici).

La précondition de l'algorithme est que la liste des arbres contient au moins un élément.

Le processus de construction se termine quand la liste contient exactement un élément, et c'est l'arbre que vous renvoyez.

Sinon, il faut identifier les deux arbres minimaux a_1 et a_2 (c'est à dire, avec la probabilité cumulée minimale), les supprimer de la liste et les remplacer par la combinaison de a_1 et a_2 (voir la fonction `combine_two`), et ensuite continuer la construction avec la nouvelle liste.

Plusieurs implantations sont possibles (itérative ou récursive, différentes manières d'identification des arbres minimaux). Nous suggérons de procéder comme suit:

- Vous avez initialement une liste contenant des arbres dont les poids sont distribués arbitrairement, par exemple:

0.2	0.15	0.3	0.25	0.1

- Triez cette liste. Si `trs` est la liste des arbres, vous obtenez une liste des arbres triés par poids (en ordre croissant) avec `sorted(trs, key=lambda x: x.val)`

0.1	0.15	0.2	0.25	0.3

- Vous pouvez maintenant combiner les deux premiers éléments et ensuite relancer l'itération:

0.25	0.2	0.25	0.3

Exercice 11 En combinant les fonctions `init_treeset` et `combine_all_trees`, il est facile d'écrire la fonction `construct_huffman_tree` qui prend une distribution de caractères comme définie dans la section 2 et construit l'arbre de Huffman correspondant.

Exercice 12 Testez votre fonction de construction avec la fonction `test_construct_huffman_tree` dans le fichier `huffman.py`.

4 Codage et décodage de textes [60 min]

4.1 Construction d'une table de codage

Pour pouvoir coder des textes, nous construisons une table de codage, selon la procédure décrite en cours.

Exercice 13 Écrivez la fonction `tab_cod` qui effectue un parcours récursif d'un arbre de Huffman pour construire une table de codage.

Vous pouvez vous inspirer très fortement de la fonction `tab_cod` des transparents du cours. Cette fonction prend un mot `m` (listes de bits: nombres 0 et 1) et un arbre de Huffman, et renvoie une table de codage, qui sera représentée par un dictionnaire qui associe des mots à des caractères. Voir l'exercice 14 pour un exemple.

Le mot `m` représente le chemin parcouru dans l'arbre pour arriver au noeud actuel. En descendant dans le sous-arbre gauche (respectivement droit), il faut étendre ce mot avec 0 (respectivement 1). Pour ce faire, utilisez la fonction + qui permet de concaténer deux listes. (Vous ne pouvez pas écrire quelque chose comme `[0, 1] + 0` - que faut-il faire à la place?)

De même, pour la combinaison de deux dictionnaires (écrite ∪ sur les transparents), utilisez la fonction `dict_merge_f`, qui a l'effet illustré par l'exemple suivant:

```
>>> dict_merge_f ({'a': [0, 0], 'b': [0, 0, 1]}, {'c': [0, 0, 0], 'd': [1, 1, 1, 1]})
{'a': [0, 0], 'c': [0, 0, 0], 'b': [0, 0, 1], 'd': [1, 1, 1, 1]}
```

Remarque (style fonctionnel): Quelques fonctions prédéfinies en Python ont des effets de bord: en appelant ces fonctions, vous modifiez l'argument de la fonction. Ceci est illustré par la situation suivante:

```
>>> m = [0, 1, 0]
>>> m.append(1)
>>> m
[0, 1, 0, 1]
```

Dans un style de programmation purement fonctionnel, vous évitez ce comportement, qui est souvent indésirable dans des fonctions récursives. Les fonctions + et `dict_merge_f` sont des variantes fonctionnelles des fonctions prédéfinies `append` et `update`. Comparez avec la situation décrite plus haut:

```
>>> m = [0, 1, 0]
>>> m + [1]
[0, 1, 0, 1]
>>> m
[0, 1, 0]
```

Exercice 14 Écrivez la fonction `htree_to_coding_tab` qui appelle `tab_cod` avec la bonne initialisation. Vous devez ensuite la tester.

Exemple:

```
>>> ht = Node (1.0, Leaf (0.5, 'a'), Node (0.5, Leaf(0.3, 'b'), Leaf(0.2, 'c')))
>>> htree_to_coding_tab(ht)
{'a': [0], 'c': [1, 1], 'b': [1, 0]}
```

4.2 Codage d'un texte

Le but du codage est de convertir une chaîne de caractères en une chaîne de bits (ceux-ci sont représentés par des nombres 0 et 1, comme avant).

Exercice 15 Écrivez la fonction `code_source_string_to_bit_list` qui prend une chaîne de caractères (typiquement le contenu d'un fichier) et une table de codage (le résultat de toutes les constructions précédentes) et construit une liste de bits qui codent la chaîne d'entrée.

La fonction est facile à écrire, avec une itération sur les caractères de la chaîne d'entrée, et en utilisant l'instruction `extend` qui attache une liste à la fin d'une liste:

```
>>> m = [0, 1]
>>> m.extend([1, 1])
>>> m
[0, 1, 1, 1]
```

Exercice 16 Testez le codage d'un texte (`alice.txt` ou `miserables.txt`) en combinant les opérations programmées jusqu'à maintenant: lecture d'un fichier de caractères; calcul de la distribution des caractères; construction d'un arbre de Huffman et de la table de codage correspondante; codage de la chaîne de caractères avec cette table; écriture du code résultant sur fichier. La séquence des appels est suggérée dans `huffman.py`.

Quel est le taux de compression que vous obtenez?

4.3 Décodage d'un texte

A l'inverse, vous pouvez décoder un texte que vous venez de coder.

Exercice 17 Le but de cet exercice est d'écrire la fonction `decode_bit_list_to_char` qui décode un seul caractère, étant donné une liste de bits `bl`, un arbre de codage `ht`, et la position actuelle `pos` dans la liste.

La fonction parcourt récursivement l'arbre `ht`:

- Si on a atteint une feuille, le caractère à renvoyer est juste le code stocké dans la feuille.
- Si on est dans un noeud, en fonction du bit que l'on trouve dans `bl` à la position `pos`, on continue à parcourir le sous-arbre gauche ou droit, avec la position suivante.

Exercice 18 Avec des appels successifs de la fonction `decode_bit_list_to_char`, vous pouvez maintenant écrire la fonction `decode_bit_list_to_string` qui décode une liste de bits, étant donné un arbre de décodage, et qui renvoie toute la chaîne de caractères décodée.

Exercice 19 Vous pouvez maintenant décoder le texte codé que vous avez écrit précédemment sur fichier: On lit le texte, le décode et écrit la chaîne de caractères résultante sur fichier. La séquence des appels se trouve dans `huffman.py`. Si le programme est sans erreur, le texte original et le texte obtenu après décodage sont identiques.

Lors du décodage, nous avons triché à deux endroits:

- pour effectuer le décodage, nous avons utilisé l'arbre de Huffman construit précédemment, mais si vous avez uniquement connaissance du texte codé, vous ne pouvez pas l'inférer;
- de même, nous avons utilisé la taille exacte (nombre de bits) du texte code, qui lui aussi n'est stocké nulle part.

Exercice 20 *(Exercice optionnel)* Concevez un mécanisme pour stocker cette information (arbre de Huffman et taille du code), dans le même ou un autre fichier. Vous pourrez alors échanger le texte codé et cette méta-information avec vos collègues et ainsi décoder l'information produite par un autre programme de codage.

A Arbres en Python

A.1 Construction d'arbres

En cours, nous avons utilisé la notation suivante:

- `L` pour la construction de feuilles
- `N` pour la construction de noeuds

En Python, nous utilisons les constructeurs

- `Leaf` pour des feuilles
- `Node` pour des noeuds

L'écriture `Node(0.2, Leaf(0.3, 'x'), Leaf(0.4, 'y'))` en Python correspond à l'écriture `N(0.2, L(0.3, x), L(0.4, y))` vue en cours.

A.2 Parcours et décomposition d'arbres

En cours, nous avons utilisé un notation fonctionnelle pour accéder aux composants d'un arbre, par exemple:

- `L(v, c) = v` pour sélectionner la valeur numérique d'une feuille
- `N(v, a1, a2) = v` pour sélectionner la valeur numérique d'un noeud

Comme vu en cours, la somme des valeurs numériques d'un arbre s'écrit alors:

```
somme(L(v, c)) = v
somme(N(v, a1, a2)) = v + somme(a1) + somme(a2)
```

En Python, nous utilisons un schéma fixe pour sélectionner le cas d'une feuille ou d'un noeud. Soit `t` un arbre:

```
if isinstance(t, Leaf):
   ...  # traitement si t est une feuille
else:
   ...  # traitement si t est un noeud
```

Pour un arbre, on sélectionne

- la valeur numérique avec `t.val` (pour des feuilles ou noeuds)
- le caractère d'une feuille avec `t.code`
- le sous-arbre gauche respectivement droit d'un noeud avec `t.low` respectivement `t.high`

Avec ceci, la fonction `somme` s'écrit:

```
def somme (ht):
    if isinstance(ht, Leaf):
        return(ht.val)
    else:
        return(ht.val + somme(ht.low) + somme(ht.high))
```

B Les paquetages en Python

Le paquetage est un mécanisme de structuration du code qui

- le rend plus lisible: des programmes comportant possiblement des centaines ou milliers de lignes de code sont répartis sur plusieurs fichiers et peuvent être lu plus facilement
- le rend réutilisable: un paquetage peut être importé par différents programmes, et il n'est pas nécessaire de le copier physiquement.

La réutilisation se fait à l'aide d'une déclaration d'import, de la form `from` *`fichier`* `import` *`fonctions`*, où *fonctions* est une liste de noms de fonction, ou `*` pour "toutes les fonctions".

Les paquetages (fichiers `*.py`) utilisés dans le projet sont:

- `arbres`: fonctions élémentaires sur les arbres que vous devez écrire
- `bitstring`: conversion de chaînes de caractères vers des listes de bits, et inversement. Vous n'avez pas besoin de modifier ce fichier.
- `htree_construct`: votre fichier de travail principal; presque toutes les fonctions du projet doivent être programmées dans ce fichier.
- `htree_dot`: conversion d'arbres de Huffman vers le format `dot`, pour affichage.
- `htrees`: le type des arbres de Huffman, avec les définitions des constructeurs `Leaf` et `Node`.
- `huffman`: La procédure principale du projet.

Pour résumer: les fichiers avec lesquels vous travaillez sont:

- `arbres.py` pour les exercices de § 1
- `htree_construct.py` pour le reste du projet; éventuellement `huffman.py` pour des tests.

C Repl.it

Vous pouvez faire une grande partie du projet sur `repl.it`, avec la restriction que vous ne pouvez pas correctement visualiser les fichiers qui seront écrits lors du codage / décodage. Une difficulté supplémentaire vient du fait que `repl.it` ne semble pas offrir un support approprié pour le développement d'un projet consistant de plusieurs fichiers.

Voici comment procéder, pour la première partie du projet décrite dans la § 1.

- Allez sur https://repl.it/
- Cliquez sur `NewRepl` pour créer un nouveau projet; sélectionnez le langage Python 3.
- Téléversez les fichiers que vous avez extraits de l'archive `huffman.tar`. Pour cela, cliquez sur le nuage ("upload") et sélectionnez les fichiers de l'archive. Repl.it affiche des erreurs (fonctions non définies etc.) qui résultent d'un problème de gestion interne à Repl.it.
- Travaillez dans l'onglet `main.py`, où vous mettez le code suivant:

```
from arbres import *
display_htree(a1)
```

- Exécutez le code dans l'éditeur de Repl.it comme d'habitude (clic sur la flèche) pour voir le résultat dans la console de Repl.it.