TP 2 : Implantation de l'algorithme LZW

1 Introduction

Le but de ce TP est d'implanter l'algorithme LZW de compression d'un texte à l'aide d'un dictionnaire. L'algorithme de décompression correspondant est fourni.

Avant de commencer, récupérez l'archive `lzw.tar` de Moodle. Cette archive contient les fichiers :
- `lzw.py` : le fichier de travail
- `my_zip.py` et `my_unzip.py` : votre version des utilitaires `zip` et `unzip`.
- `alice.txt` : un fichier texte à utiliser pour tester votre implantation.

Nous vous demandons le travail suivant :
- implantation de l'algorithme de compression (fonction `compress`), version simple (§ 2) ;
- modification de cet algorithme, pour des dictionnaires de taille limitée (§ 3) ;
- expérimentation avec plusieurs tailles de dictionnaires (§ 4).

Si vous avez envie d'aller plus loin, nous vous donnons quelques pistes dans la § 5.

2 Compression simple

Ouvrez le fichier `lzw.py`. Vous voyez que trois constantes globales sont prédéfinies :
- `alpha_small`, indiquant si vous voulez travailler avec des "petits" ou "gros" alphabets.
- `alpha_size`, la taille de l'alphabet qui sera utilisé.
- `dict_size`, la taille maximale du dictionnaire qui sera construit. Vous pouvez négliger cette constante pour les exercices de cette section ; elle jouera un rôle dans les exercices de la § 3.

Dans les exercices de cette section, nous travaillerons avec des petits alphabets. Mettez donc :

```
alpha_small = True
alpha_size = 4
```

Avec ceci, vous pourrez traiter des chaînes de caractères qui contiennent des lettres `a,b,c,d`.

Vous savez qu'un élément clé de l'algorithme LZW est la construction d'un dictionnaire. Nos dictionnaires seront des dictionnaires au sens de Python. Référez-vous au TP 1 pour un rappel des instructions essentielles.

Exercice 1 Les dictionnaires construits pour tester l'implantation de la fonction de compression sont petits, de la forme `{'a': 0, 'b': 1 ...}` pour un petit nombre de caractères (peut-être 3 ou 5). Écrivez la fonction `init_dict` qui renvoie un tel dictionnaire. Notez que la fonction est sans paramètres, mais dépend de la constante globale `alpha_size`.

Exemple :

```
>>> alpha_size = 4
>>> init_dict()
{'a': 0, 'c': 2, 'b': 1, 'd': 3}
```

Note : La fonction `ord` renvoie la position d'un caractère dans l'alphabet ; son inverse, la fonction `chr`, renvoie le caractère qui a la position donnée dans l'alphabet. Par exemple, `ord('a')` est 97 et `chr(100)` est `'d'`.

Exercice 2 Écrivez la fonction `compress` qui prend une chaîne de caractères `st` et renvoie une liste de nombres (positions dans le dictionnaire) qui est le code du texte compressé. En même temps que cette liste, la fonction construit un dictionnaire, qui n'est pas renvoyé par la fonction.

Pour l'implantation de cette fonction, inspirez-vous de l'algorithme sur les transparents du cours.

Votre fonction doit s'exécuter correctement pour n'importe quelle `alpha_size` positive. Il doit notamment être possible de modifier la définition de `alpha_size` au début du fichier pour des alphabets plus larges sans adapter le code de la fonction `compress` ; mais voir aussi l'exercice 6.

Note : Il peut être instructif d'insérer des `print` à l'intérieur de la boucle principale de l'algorithme. Prenez soin de commenter ces instructions pour la version finale de l'algorithme, qui ne doit pas afficher des traces de l'exécution de l'algorithme.

Exercice 3 Testez votre fonction avec les exemples vus en cours-TD.

Exercice 4 Avant de tester la fonction de décompression, écrivez la fonction `init_inv_dict` qui construit le dictionnaire inversé (par rapport à la fonction `init_dict`).

Exemple :

```
>>> alpha_size = 4
>>> init_inv_dict()
{0: 'a', 1: 'b', 2: 'c', 3: 'd'}
```

Exercice 5 Vous pouvez maintenant tester la fonction `decompress` et constater qu'elle est l'inverse de la fonction `compress` :

```
>>> compress("abaabcabadaba")
[0, 1, 0, 4, 2, 4, 0, 3, 9]
>>> decompress([0, 1, 0, 4, 2, 4, 0, 3, 9])
'abaabcabadaba'
```

3 Compression avec un dictionnaire à taille limitée

On va désormais utiliser des alphabets plus grands. Pour les exercices suivants, redéfinissez `alpha_size = 128` et mettez `alpha_small = False`. Nous rappelons que le code ASCII comporte 128 caractères, et que ce sont les mêmes que les 128 premiers caractères du code UTF-8.

Exercice 6 Modifiez les fonctions `init_dict` et `init_inv_dict` (par une simple distinction de cas) pour que les dictionnaires ne commencent pas avec le caractère `'a'`, mais avec le caractère 0 du code ASCII si `alpha_small` est `False`. Vous pourrez donc désormais facilement basculer entre les deux modes (petits et gros dictionnaires).

```
>>> init_dict()
{'\x00': 0, ...,  'a': 97, 'b': 98,  ...,  '\x7f': 127}
>>> init_inv_dict()
{0: '\x00', ...,  97: 'a', 98: 'b',  ...,  127: '\x7f'}
```

Exercice 7 Testez la fonction de compression sur un texte plus large, par exemple `alice.txt` :

```
>>> compress(read_file("alice.txt"))
```

Quelle est la taille (en nombre de caractères) du texte original ? Et la taille (nombre de codes) du texte compressé ? Quelle est la taille (nombre d'entrées) du dictionnaire construit ?

Vous voyez surtout que les codes (indices du dictionnaire) du texte compressé ne peuvent pas être représentés avec un seul octet. Pour ce faire, nous allons limiter la taille du dictionnaire à 256, avec la définition `dict_size = 256` et une modification de l'algorithme de compression.

Exercice 8 Écrivez la fonction `compress_lim` qui limite la taille du dictionnaire construit à `dict_size`. Cette fonction diffère de la fonction `compress` seulement dans un détail : elle ne rajoute pas d'autre élément au dictionnaire si celui est plein.

Notes :

— Pour écrire `compress_lim`, copiez la fonction `compress` et effectuez la modification dont il est question.

— Encore une fois, votre fonction doit s'exécuter correctement pour n'importe quelle valeur raisonnable de `dict_size`. « Raisonnable » veut surtout dire : la taille du dictionnaire doit être supérieure ou égale à la taille de l'alphabet.

4 Expérimentation

Avec les codes du texte compressé limités aux valeurs $0 \ldots 255$, il sera possible de construire une chaîne de caractères qui peut être écrite dans un fichier.

Exercice 9 Écrivez la fonction `compressed_code_to_string` qui prend une séquence de codes (compris entre 0 et 255) et produit la chaîne de caractères correspondante (utilisez la fonction `chr`).

Exercice 10 Écrivez aussi la fonction inverse `string_to_compressed_code` qui sera utilisée pour récupérer le code compressé d'un fichier.

Après avoir écrit toutes ces fonctions, vous pouvez les utiliser comme scripts sur la ligne de commande de votre console. Le code dans les fichiers `my_zip.py` et `my_unzip.py` montre comment. Vous pouvez survoler rapidement le code de `my_zip.py` sans essayer de comprendre tous les détails. Avec `sys.argv[1]`, vous récupérez le premier argument du script (pareil pour `sys.argv[2]`), vous lisez le code dans le fichier nommé par le premier argument, ensuite vous compressez le contenu de ce fichier et l'écrivez dans le fichier nommé par le deuxième argument. La structure de `my_unzip.py` est similaire.

Voici comment procéder, dans une console (et non dans l'interpréteur de Python) :

— Assurez-vous que les deux fichiers sont bien exécutables :

```
> ls -la
...
-rwxr-xr-x 1 nom users  338 Mar 20 00:54 my_zip.py
-rwxr-xr-x 1 nom users  354 Mar 20 00:54 my_unzip.py
...
```

La ligne doit commencer avec `-rwx` pour montrer que le fichier est exécutable. Si ce n'est pas le cas, tapez

```
> chmod +x my_zip.py
> chmod +x my_unzip.py
```

— Vous pouvez compresser un ficher, par exemple `alice.txt`, de la manière suivante :

```
> ./my_zip.py alice.txt alice_compr.txt
```

— et ensuite le décompresser avec

```
> ./my_unzip.py alice_compr.txt alice_decompr.txt
```

— Vérifiez que l'original et le texte décompressé sont les mêmes :

```
> diff alice.txt alice_decompr.txt
>
```

Si aucune différence n'est affichée, votre programme fonctionne correctement.

— Vous pouvez afficher la taille des fichiers avec `ls -la`

Exercice 11 Envoyez un texte comprimé à un collègue et demandez-lui de le décompresser avec son algorithme. Il devrait obtenir votre texte original.

5 Pour aller plus loin

Comparez le taux de compression de votre programme avec celui de `gzip`[1] :

```
> gzip alice.txt
```

Contrairement à notre programme, `gzip` remplace l'original par le fichier compressé. Pour retrouver l'original, faites

```
> gunzip alice.txt.gz
```

Vous constatez que nos fonctions `compress` et `decompress` ne sont pas très performants[2] par rapport à `gzip` et `gunzip`.

Voici quelques pistes de réflexion :

1. Comparez la taille du dictionnaire non borné (exercice 2) avec 256, et vous voyez que le dictionnaire limité à 256 entrées est trop petit. Soyez généreux et admettez une taille du dictionnaire de $256 \times 256 = 65536$, il vous faut donc deux octets pour représenter une entrée du dictionnaire. Réécrivez la fonction `compressed_code_to_string`, et testez votre algorithme de compression.
2. Le résultat est décevant, parce que maintenant, le dictionnaire peut devenir trop grand. Au lieu de représenter un code (= une entrée de dictionnaire) par un octet ou deux octets, une solution intermédiaire pourrait être appropriée, par exemple 13 bits, en fonction de la taille du dictionnaire non bornée construit lors de la compression. Écrivez une fonction qui ressemble à `compressed_code_to_string` (exercice 9). Cette fonction n'est pas facile à écrire, parce que la séquence de bits que vous obtenez n'est pas un multiple de 8, donc difficile à coder comme séquence d'octets.
3. Une autre piste consiste à combiner l'algorithme LZW avec l'algorithme de Huffman : Vous voyez que les codes du texte compressé n'ont pas la même fréquence. Vous pouvez donc établir une distribution de fréquence des codes du texte comprimé par LZW, qui est la base pour une compression à la Huffman, mais cette fois-ci avec des codes du texte compressé et non avec des caractères du texte original.

1. https://www.gnu.org/software/gzip/
2. et ceci tient aussi pour la complexité algorithmique ; les programmes ne sont pas aptes à compresser de grandes quantités de données parce que le traitement des chaînes de caractères n'est pas optimisé.