

LOG3430 – Méthodes de test et de validation du logiciel

Laboratoire 3
Tests OO – MaDUM

Département de génie informatique et de génie logiciel
École Polytechnique de Montréal



Soumis par
Roman Zhornytskiy (1899786)
Hakim Payman (1938609)
Gabriel Tagliabracci (1935775)

Groupe : 02

Soumis à Nouredine Kerzazi

Hiver 2020

4. Travail à effectuer

4.1. Identifier la situation la plus favorable pour un test MaDUM ensuite construire le MaDUM en identifiant respectivement les constructors, reporters, transformers, et autres pour les attributs de la classe `huffman.py`.

Le cas idéal pour un test MaDUM est un cas tel qu'il implique une quantité minimale de séquences d'opérations à exécuter pour réaliser les tests d'une tranche. Pour le MaDUM présent, nous avons omis d'inclure les méthodes « helpers » (`__eq__`, `__lt__`, `__repr__` et `isLeaf`).

Tableau 1: MaDUM de la classe `Huffman`

	<code>__init__</code>	<code>build_codebook</code>	<code>from_string</code>	<code>encode_tree</code>	<code>unzip_tree</code>	<code>huffman_encode</code>	<code>huffman_decode</code>
<code>weight</code>	C		O				
<code>data</code>	C	O	O	O	O		O
<code>left</code>	C	O		O			O
<code>right</code>	C	O		O			O
<code>codebook</code>	C	T				O	

4.2. À l'aide de `unittest`, écrire une classe de test unitaire pour tester les tranches identifiées dans l'étape précédente. Pour chaque tranche, la séquence des méthodes doit suivre le principe de MaDUM.

Nous avons testé les séquences d'opérations pouvant être déterminées dans chacune des tranches du MaDUM précédemment fait. Les tests pour chacune des tranches ont été écrits dans le fichier `test_huffman.py`.

4.3. Est-ce que cet exemple montre une ou plusieurs limitations de MaDUM ? Si oui, pouvez-vous les citer ?

Oui, il y a quand même des combinaisons d'opérations qui nous échappent en raison de l'omission des méthodes « helpers ». De plus, on constate aussi que la gestion des erreurs dans ce cas n'a pas pu être couverte avec MaDUM.

4.4. À l'aide de l'outil Coverage.py, évaluez la couverture de la classe Huffman.py (dans le fichier huffman.py) et identifiez les parties de code non couvertes, s'il y en a. Pour les parties non couvertes, essayez de faire des tests boîte blanche pour atteindre la couverture maximale.

Coverage.py nous a donné le résultat suivant :

Name	Stmts	Miss	Cover	Missing
huffman.py	138	6	96%	54-56, 137, 182, 210
test_huffman.py	100	0	100%	
TOTAL	238	6	97%	

Figure 1: Couverture lors de la première exécution de Coverage.py

Il nous faut alors faire les tests pour les méthodes se trouvant aux lignes 54 à 56, 137, 182 et 210 (Voir le fichier test_huffman.py). Voici la couverture maximale que nous avons pu atteindre suite à la complétion des tests :

Name	Stmts	Miss	Cover	Missing
huffman.py	138	0	100%	
test_huffman.py	123	0	100%	
TOTAL	261	0	100%	

Figure 2: Couverture suite à la complétion des tests

4.5. Qu'est-ce qui se passe si la fonction responsable du padding ne fonctionne pas comme attendu ? Proposez une solution pour corriger le bogue.

Si la fonction responsable d'ajouter le padding (*compress_binary_string*) ne fonctionne pas comme prévu, alors il sera impossible de bien décompresser un string qui a été compressé, car la décompression dépend de la justesse de l'ajout du padding. Ainsi, un padding qui a été mal ajouté à un string, résulte inévitablement en une décompression qui échoue.

4.6. Complétez le cas de test suivant pour couvrir des cas extrêmes.

```
class TestZip(unittest.TestCase):  
    def test_collectionStrings(self):  
        collectionStrings = {  
            'ABCDEFGHIIJKLMNOPQRSTUVWXYZ0123456789',  
            'XXXX-XXXX-XXXX-XXXX',  
            '*KWxx5byy12hri3l3lKRAB',  
            '18NPLdeep',  
            'Gsssssssssss1111111111111111*****',  
            'AAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCBbbbbbbbbb',  
            'ooooooooooooooooooooooooooooooooooooooooooooooooooooo',  
            '\n{a-9}~\\x^&`<>^^2*_MM|grep@||\`',  
        }  
        ...
```

Cas de Test

S : string

S1 -> t1 = < {S = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"}, {A: 000000, B: 000001, C: 000010, D: 000011, E: 000100, F: 000101, G: 000110, H: 000111, I: 001000, J: 001001, K: 001010, L: 001011, M: 001100, N: 001101, O: 001110, P: 001111, Q: 010000, R: 010001, S: 010010, T: 010011, U: 010100, V: 010101, W: 010110, X: 010111, Y: 011000, Z: 011001, 0: 011010, 1: 011011, 2: 011100, 3: 011101, 4: 011110, 5: 011111, 6: 100, 7: 101, 8: 110, 9: 111} >

S2 -> t2 = < {S = "XXXX-XXXX-XXXX-XXXX"}, {X: 0, -: 3} >

S3 -> t3 = < {S = "KWxx5byy12hri3l3KRAB}, {K: 0000, W: 0001, x: 0010, 5: 0011, b: 01000, y: 01001, 1: 01010, 2: 01011, h: 011, r: 1000, i: 1001, 3: 1010, l: 1011, R: 1100, A: 1101, B: 1110, *: 1111} >

S4 -> t4 = < {S = "18NPLdeep"}, {1: 0000, 8: 0001, N: 0010, P: 0011, L: 0100, d: 0101, e: 011, p: 1} >

```
S5 -> t5 = < {S = Gsssssssss1111111111111111*****}, {G: 000, s: 001, 1: 01,  
*: 1}>
```

S6 -> t6 = < {S = "AAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBB"},
{B: 00, C: 01, A:1} >

S7 -> t7 = < {S = "QQ"},
{Q: 0} >

S8 -> t8 = < {S = "\n{a-9}~\x^&`<>^^2*_MM|grep@||\"}, {\n: 00000, {: 00001, a: 00010,
-: 00011, 9: 00100, }: 00101, ~: 00110, x: 00111, &: 01000, `: 01001, <: 01010, >: 01011, 2:
01100, *: 01101, _: 01110, g: 01111, r: 10000, e: 10001, p: 10010, @: 10011, M: 1010, ^:
1011, \: 110, |: 111} >