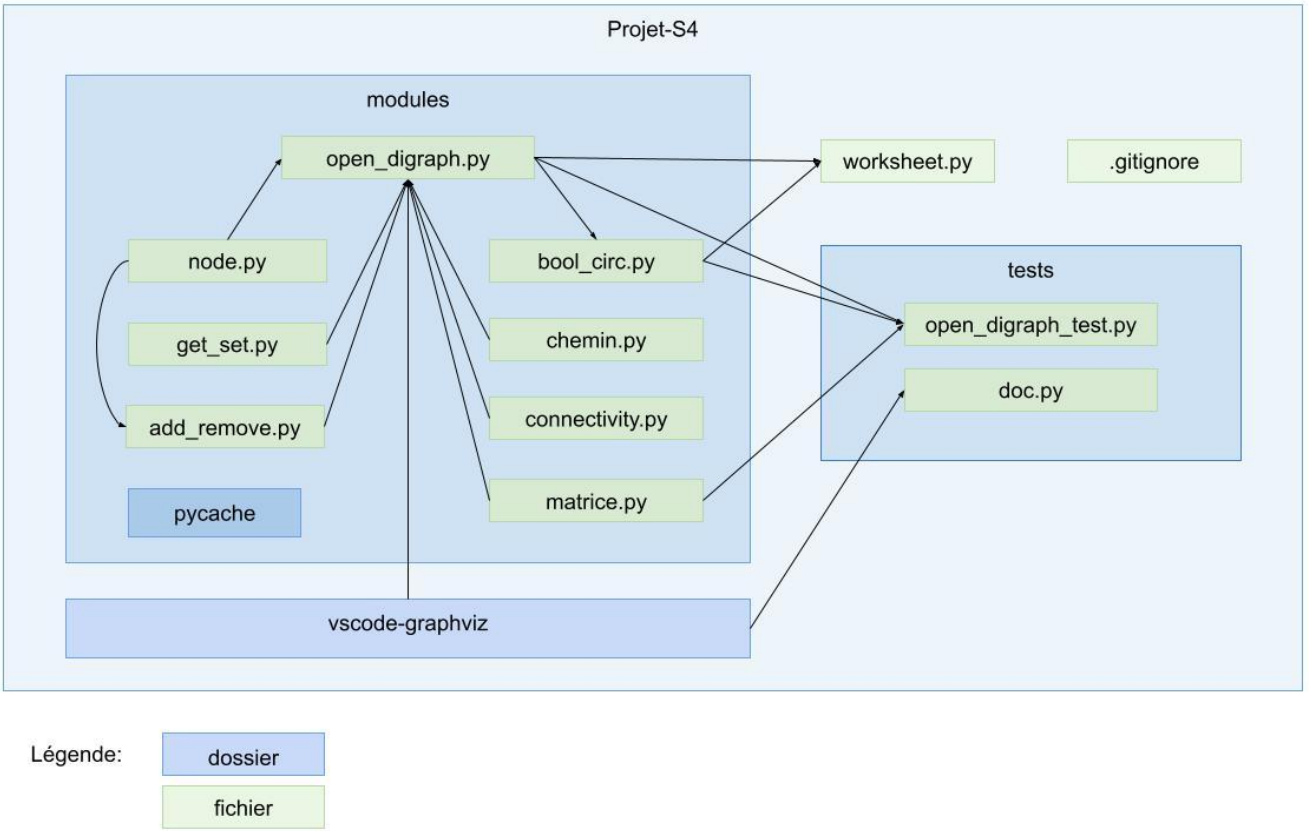
Rapport de projet d'informatique

I. Architecture du projet

Le schéma ci-dessous récapitule notre projet, c'est-à-dire les dossiers et les fichiers qui le composent ainsi que les dépendances entre les fichiers. Pour plus de clarté, nous n'avons pas explicité le contenu des dossiers dont nous ne sommes pas les auteurs.



Note concernant les tests: Au cours de ce projet, nous avons testé nos fonctions systématiquement et de diverses manières selon notre avancée. Au début nous avons principalement effectué les tests dans le fichier 'open_digraph_test.py' à l'aide du module de tests. Nous avons également vérifié nos tests à l'aide d'affichage dans le terminal depuis les fichiers 'open_digraph_test.py' et 'worksheet.py'. Une fois que l'on a pu générer des fichiers '.dot' et les afficher en PDF, nous avons principalement utilisé cet outil pour vérifier les fonctions sur les circuits booléens car cela nous permettait de comprendre de manière plus intuitive ce qu'il se passait.

II. Half-adder

Nous avons implémenté les Adder et Half-Adder qui sont utilisés afin de calculer la somme de deux entiers de taille 2^n.

Exemple d'évaluation d'un half-adder :

Voici un exemple de calcul simple, l'addition de 2 et 1 de taille 2 grâce à la fonction ci-dessous.

```
def calcul(a,b,taille) :
    '''
    input:
        int: premier entier
        int: deuxieme entier
        int: taille du registre

    output: bool_circ

    Fonction qui calcul l'addition de deux entiers
    '''
```

Étape 1 : Il faut pour chaque entier le transformer en registre de taille 2.

Étape 2 : Nous créons un graphe parallèle contenant les registres à la suite et dans le bon ordre.

Étape 3 : Ainsi comme la taille = 2 = 2^1, nous allons donc utiliser un additionneur : Half-Adder1.

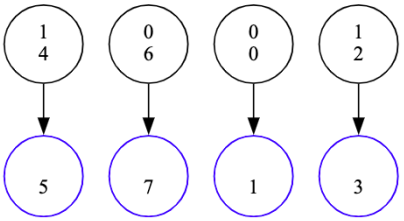
Étape 4 : On fait une composition de ces deux graphes obtenus où les entrées sont les bits correspondants au registre de nos entiers, et en sortie les bits résultats ('r') et un bit 'carry' ('cp').

Étape 5 : Appel de la fonction *'evaluate()'* qui va nous permettre d'appliquer les règles de transformations et arriver au résultat attendu.

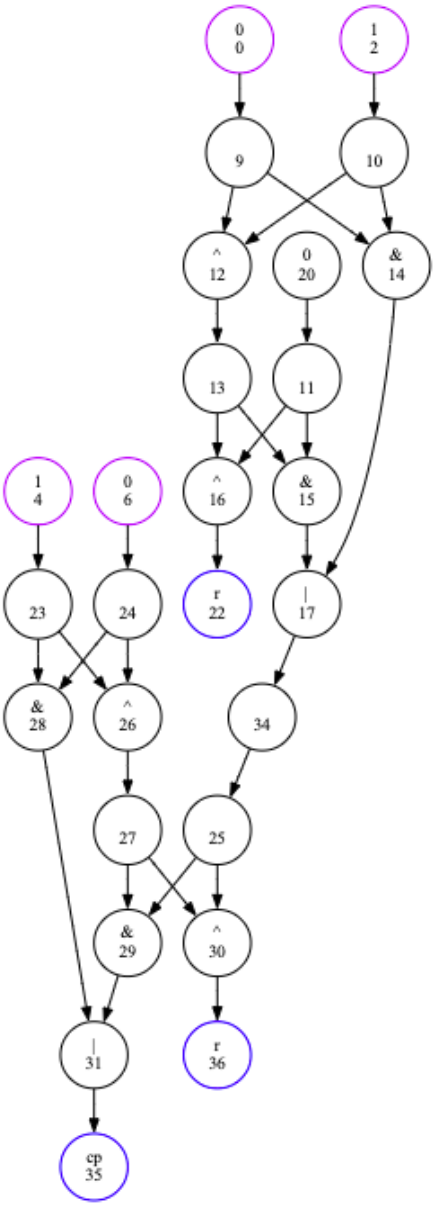
```
@classmethod
def registre(cls,n, taille) :
    '''
    inputs : int, int
    ouputs : bool_circ

    retourne un circuit booleen qui représente le registre instancié en l'entier
    '''
```

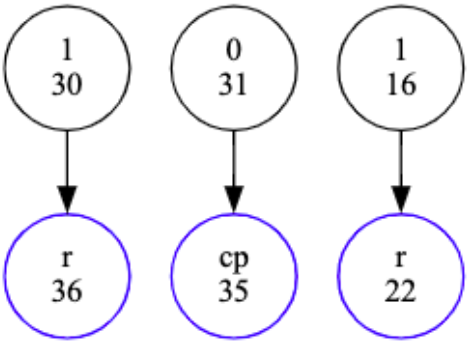
Etape 1



Etape 2



Etape 4



Etape 5, résultat final

Ainsi, on a que les bits résultats sont égaux à 1, et le bit retenu est égal à 0.
En prenant en compte que le bit de poids fort est à gauche on a que le résultat est :

$$0\ 11 = 3 = 2 + 1,$$

Propriétés d'un Half-Adder :

Etudions un peu plus en détail un élément clé de ce calcul, les Half-Adder.

1) Profondeur

On a admis que la profondeur d'un graphe vide était de 0, donc la profondeur d'un nœud est égale à 1.
Ainsi, on obtient grâce à la fonction ci-dessous, que la profondeur de Half-Adder0 est de 7.

```python
def profondeur(self, noeud=None):
    '''
    input : int ;id du noeud
    output : profondeur

    Renvoie la profondeur du graphe si le paramètre noeud est à None, sinon renvoie la profondeur du noeud
    Profondeur graphe vide : 0
    '''
```

On trouve (par un raisonnement expliqué aux lignes suivantes) :

Profondeur(Half-Adder$_n$) = 3 + 4*2^n, pour tout entier n positif

Résultats avec la formule trouvée :

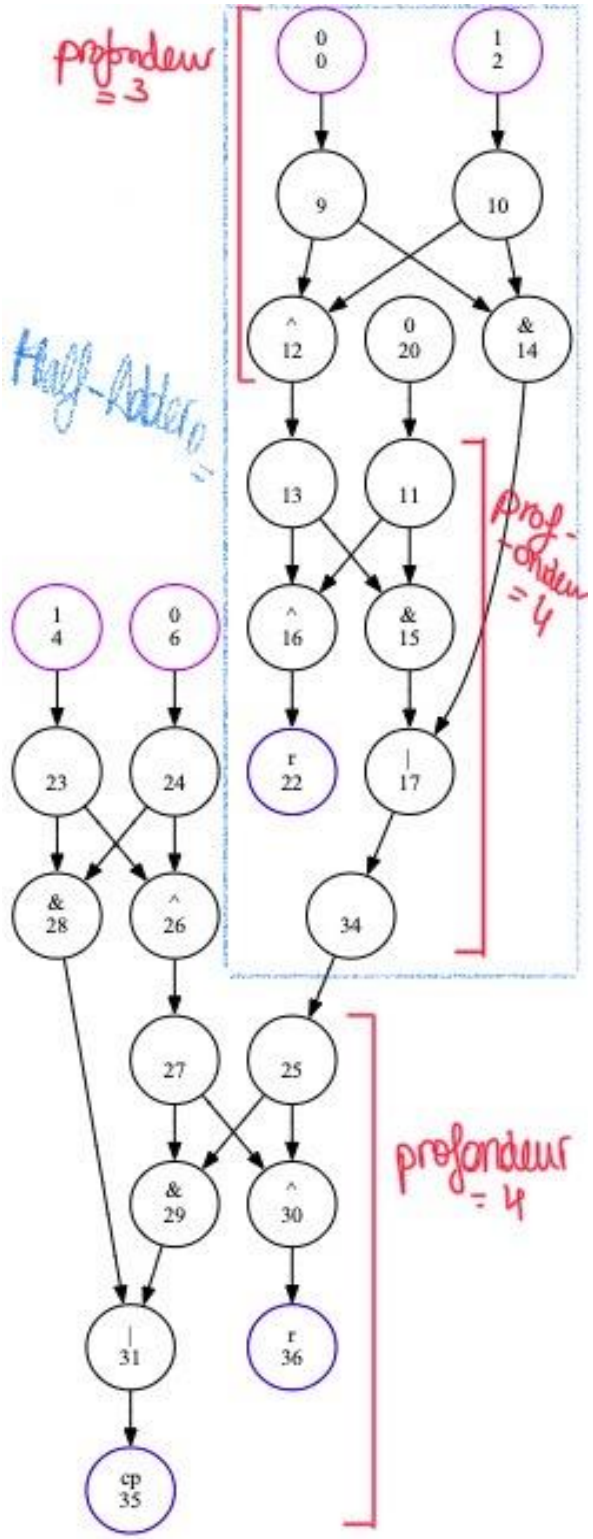
Pour n = 0 : 7
Pour n = 1 : 11
Pour n = 2 : 19
Pour n = 3 : 35
…
Pour n = 10 : 4099

Résultats numériques :

```
Profondeur d'un Half-Adder_n
n = 0: 7
n = 1: 11
n = 2: 19
n = 3: 35
n = 10: 4099
```

La formule trouvée est assez intuitive car Half-Adder$_0$ est de profondeur 7. Comme lorsque l'on calcule un Half-Adder de taille N, on combine deux Half-Adder de taille N-1. Ainsi, pour un Half-Adder$_N$ est constitué de 2^N Half-Adder$_0$ où les bits retenus sont liés deux à deux.. Ainsi, nous avons 4 nœuds qui séparent nos Half-Adder à chaque fois. Alors, on multiplie ce nombre de nœuds par le nombre de Half-Adder$_0$ qu'il y a au total et on rajoute les 3 premiers nœuds de notre Half-Adder$_n$. (3+4 = 7)

Voici un schéma explicatif avec Half-Adder1 constitué de 2^1 = 2 Half-Adder0 :



2) Nombre de portes

De même, pour le nombre de portes, on a que pour un Half-Adder$_0$ le nombre de portes est de 5.
Comme lorsque que l'on crée un Half-Adder$_n$ alors on assemble deux Half-Adder $_{n-1}$.
On a, sous forme récurrente, pour n > 0:

$$NbPortes(Half\text{-}Adder_n) = 2*NbPortes(Half\text{-}Adder_n),$$
$$\text{avec } NbPortes(Half\text{-}Adder_0) = 5$$

Ainsi, en passant par la forme explicite, on a, pour tout entier naturel :

$$NbPortes(Half\text{-}Adder_n) = 5*2^n$$

Résultats avec la formule trouvée :
Pour n = 0 : 5
Pour n = 1 : 10
Pour n = 2 : 20
Pour n = 3 : 40
…
Pour n = 10 : 5120

Résultats numériques :

```
Nb portes d'un Half-Adder
n = 0: 5
n = 1: 10
n = 2: 20
n = 3: 40
n = 10: 5120
```

3) Longueur du plus court chemin entre une entrée et une sortie

La longueur du plus court chemin entre une entrée et une sortie est de 5 pour un Half-Adder$_0$. En effet, le chemin lie n'importe quel nœud entrée avec le nœud de sortie qui est la retenue.
Or, on remarque que la longueur du plus court chemin entre une entrée et une sortie est de 5 peu importe notre Half-Adder. En effet, dans un Half-Adder de taille n, la partie la plus inférieure de ce circuit représente un Half-Adder$_0$ ainsi la longueur du plus court chemin dans cette partie est de 5. Cependant, cette longueur est la plus petite étant donné que nous relions les nœuds représentants les bits de retenue. Il n'y a donc pas de nouveau plus court chemin lors de la création d'un Half-Adder.

Donc, pour tout entier naturel:

$$LPCC(Half\text{-}Adder_n) = 5$$

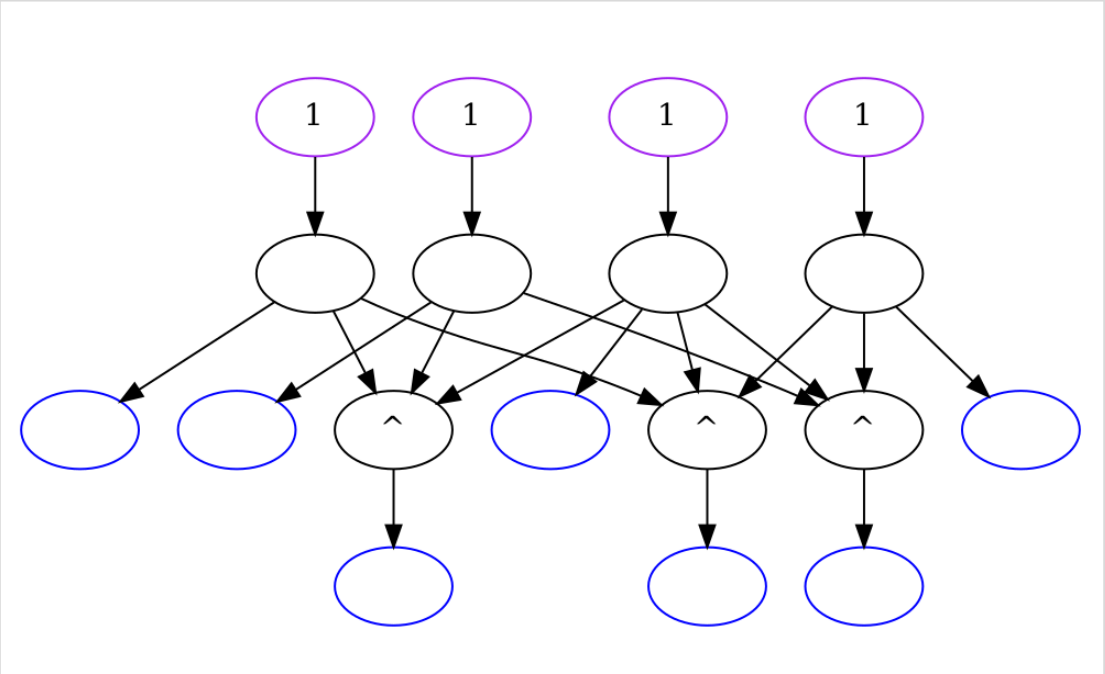
NB: LPCC = Longueur du Plus Court Chemin

Résultats numériques :

```
La longueur plus court chemin d'un Half-Adder
n = 0: 5
n = 1: 5
n = 2: 5
n = 3: 5
```

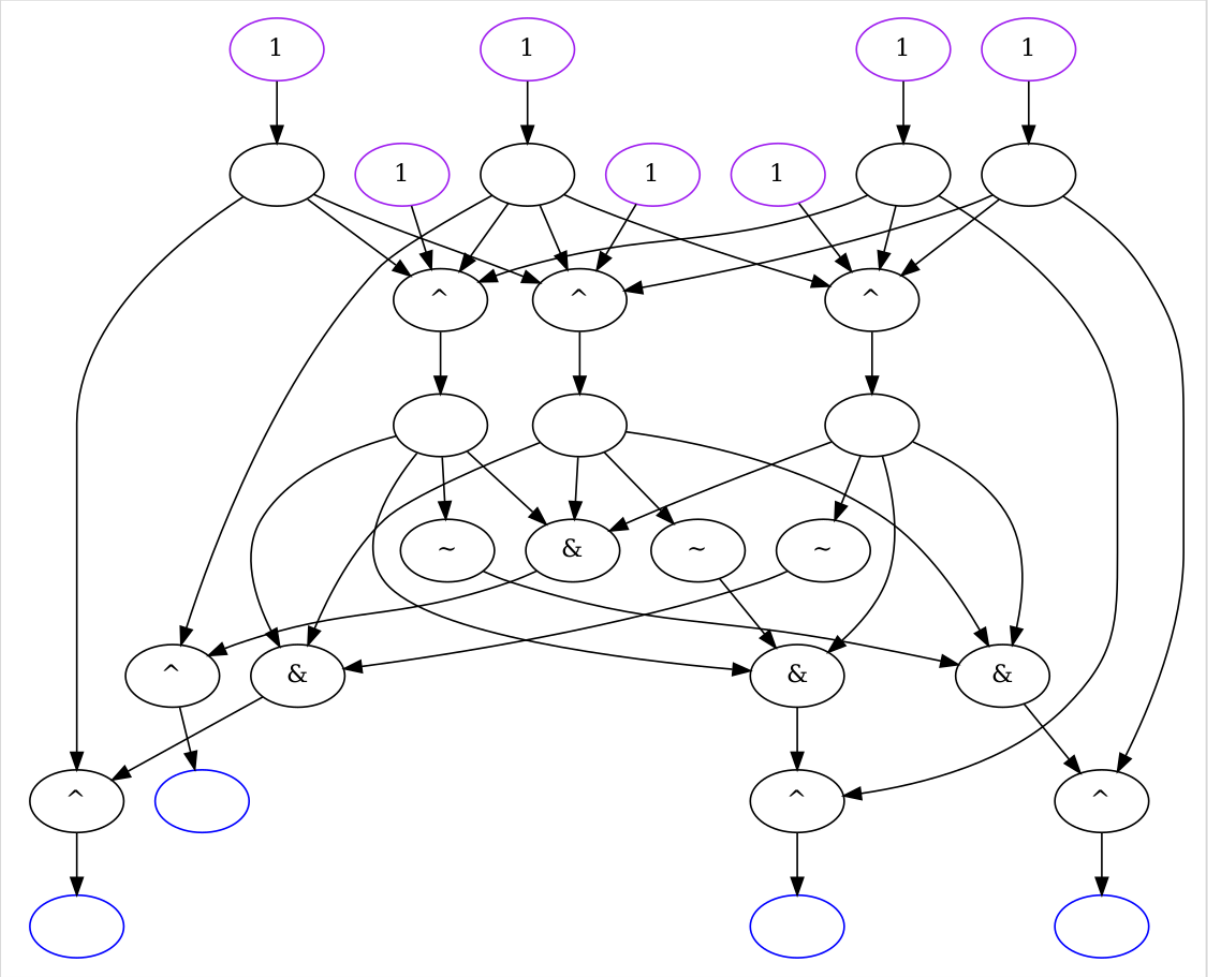
III. Code de Hamming

Pour vérifier les propriétés de code de Hamming, il a d'abord fallu coder les circuits booléens de l'encodeur et du décodeur. Voici les graphes que nous avons construit:



*Graphe de l'**encodeur** non-évalué dont les bits d'entrée valent tous 1*

NB: les noeuds violets correspondent aux entrées (ou input) de notre circuit booléen, ceux en bleu correspondent aux sorties (ou output).



*Graphe du **décodeur** non-évalué dont les bits d'entrée correspondent aux valeurs de sortie de l'encodeur ci-dessus*

a) Code de Hamming sans bruit

Afin de vérifier la propriété du code de Hamming dans l'exercice 3 du TD, on évalue le circuit de l'encodeur à l'aide d'une boucle pour toutes les combinaisons d'entrées possibles, puis on récupère les valeurs en output, et on vérifie que lorsqu'on évalue ces valeurs dans le circuit du décodeur on obtient bien le même code.

Voici le code de la fonction qui permet d'obtenir le code à 7 bits à l'aide de l'encodeur:

```
def code(b0,b1,b2,b3):
    '''
    input: int, int, int, int; valeurs des bits à encoder
    output: int (*7); code à 7 bits obtenu grâce à l'encodeur

    Fonction qui permet d'obtenir un code à 7 chiffres par l'encodeur
    '''
    encodeur=bool_circ.encodeur(b0,b1,b2,b3)
    encodeur.eval()
```

```
    res=encodeur.get_node_by_ids(encodeur.get_output_ids())
    return (int(res[0].get_label()),int(res[1].get_label()),int(res[2].get_label()),
            int(res[3].get_label()),int(res[4].get_label()),int(res[5].get_label()),
            int(res[6].get_label()))
```

Ci-dessous le code de la fonction qui permet de décoder le code à 7 chiffres à l'aide du décodeur:

```
def decode(b0,b1,b2,b3,b4,b5,b6):
    '''
    input: int (*7); valeurs des bits à décoder
    output: int, int, int, int; code à 4 bits

    Fonction qui permet de décoder un code obtenu grâce à l'encodeur
    '''
    encodeur=bool_circ.decodeur(b0,b1,b2,b3,b4,b5,b6)
    encodeur.eval()
    res=encodeur.get_node_by_ids(encodeur.get_output_ids())
    return (int(res[0].get_label()),int(res[1].get_label()),
            int(res[2].get_label()),int(res[3].get_label()))
```

Ensuite, dans le fichier test, on génère les 24 combinaisons possibles à l'aide de boucles 'for' imbriquées et on vérifie que les bits d'entrée à l'encodeur sont identiques aux bits de sortie du décodeur:

```
def test_Code_Hamming(self):
        '''
        Vérification que les valeurs d'entrée sont identiques à celles
        de sortie lorsqu'appliquées au Code de Hamming sans bruit'''
        for a in range (2):
            for b in range (2):
                for c in range (2):
                    for d in range (2):
                        b0,b1,b2,b3,b4,b5,b6=code(a,b,c,d)
                        ra, rb, rc, rd = decode(b0,b1,b2,b3,b4,b5,b6)
                        self.assertEqual(a,ra)
                        self.assertEqual(b,rb)
                        self.assertEqual(c,rc)
                        self.assertEqual(d,rd)
```

Comme l'indique le résultat lors de l'exécution du fichier test, aucune erreur n'est levée et on a ainsi vérifié que la composition de l'encodeur et du décodeur réduit à l'identité.

b) Code de Hamming avec bruit

Pour vérifier que le code de Hamming fonctionne même lorsqu'il y a un peu de bruit, c'est à dire qu'un bit change de valeur alors qu'il ne devrait pas, on peut effectuer le même test que ci-dessus mais en faisant appel à une fonction bruit qui modifie la valeur d'un des outputs de l'encodeur. Cette modification se traduit par l'ajout d'une porte 'NON' avant un des outputs.

```
    def bruit(self, o):
        '''
        input: int; indice d'un noeud en output de l'encodeur
        (on suppose la valeur comprise entre 0 et 6)

        Ajoute du bruit au code de Hamming en corrompant 1 bit sur les 7 du code
        '''
        if o<=3:
            ido=8+o
            idi=o
        else:
            ido=11+o
            idi=8+o
        no=self.get_node_by_id(11+o)
        n= node(20, '~', {idi:1}, {ido:1})
        self.add_nodes(n)
        self.add_edge(idi,20)
        self.remove_edge(idi,ido)
```

Ensuite, on crée une nouvelle fonction ‘code_bruit’ quasi-identique à celle présentée ci-dessus nommée ‘code’, la différence étant qu’on ajoute un appel à la fonction ‘bruit’ puis on évalue à nouveau:

```
    def code_bruit(b0,b1,b2,b3, bc):
        '''
        input: int, int, int, int, int; valeurs des bits à encoder
        et indice du bit avec du bruit

        output: int (*7); code à 7 bits obtenu grâce à l'encodeur

        Fonction qui permet d'obtenir un code à 7 chiffres par l'encodeur
        avec un peu de bruit (ie un bit corrompu en sortie)
        '''
        encodeur=bool_circ.encodeur(b0,b1,b2,b3)
        encodeur.eval()
        encodeur.bruit(bc)
        encodeur.eval()
        res=encodeur.get_node_by_ids(encodeur.get_output_ids())
        return (int(res[0].get_label()),int(res[1].get_label()),int(res[2].get_label()),
                int(res[3].get_label()),int(res[4].get_label()),int(res[5].get_label()),
                int(res[6].get_label()))
```

Il ne nous reste plus qu’à vérifier que les tests passent comme la fois précédente en vérifiant que le code persiste pour toutes les combinaisons possibles et peu importe où le bruit a lieu:

```
    def test_Code_Hamming_bruit_faible(self):
        '''
        Vérification que les valeurs d'entrée sont identiques à celles
        de sortie lorsqu'appliquées au Code de Hamming avec du bruit faible
        '''
        for a in range (2):
            for b in range (2):
```

```
                    for c in range (2):
                        for d in range (2):
                            for bruit in range(7):
                                b0,b1,b2,b3,b4,b5,b6=code_bruit(a,b,c,d,bruit)
                                ra, rb, rc, rd = decode(b0,b1,b2,b3,b4,b5,b6)
                                self.assertEqual(a,ra)
                                self.assertEqual(b,rb)
                                self.assertEqual(c,rc)
                                self.assertEqual(d,rd)
```

Cependant, lors de l'exécution, une erreur est levée car le bruit altère la persistance du code comme le prouve le test suivant effectué dans ‘worksheet.py’.

```
TEST du Code de Hamming avec bruit pour la valeur d'entrée 0101

On ajoute du bruit à un indice >4, par exemple 6:

Code à la sortie de l'encodeur avec le bruit:
(0, 1, 0, 1, 0, 1, 1)
Code restitué par le décodeur:
(0, 1, 0, 1)

On utilise le code sans bruit en témoin:
Code à 7 bits:
(0, 1, 0, 1, 1, 0, 0)
Code restitué par le décodeur:
(0, 1, 0, 1)
```

```
TEST du Code de Hamming avec bruit pour la valeur d'entrée 0101

On ajoute du bruit à un indice <=4, par exemple 2:

Code à la sortie de l'encodeur avec le bruit:
(0, 1, 1, 1, 0, 1, 0)
Code restitué par le décodeur:
(0, 1, 1, 1)

On utilise le code sans bruit en témoin:
Code à 7 bits:
(0, 1, 0, 1, 1, 0, 0)
Code restitué par le décodeur:
(0, 1, 0, 1)
```

On observe en effet que si on ajoute du bruit à un bit en position inférieure à 4, le bruit altère considérablement le code, alors que si le bruit se fait sur les 3 derniers indices, il n’a aucune conséquence sur le code. En regardant de plus près, on remarque que le code donné en entrée à l’encodeur correspond aux 4 premiers chiffres donnés en sortie dans le code à 7 chiffres. Cela explique donc pourquoi est-ce que le bruit altère la persistance du code.

Le résultat obtenu ne correspond pas à celui attendu donc on ne peut pas vérifier les propriétés du Code de Hamming à l’aide de nos fonctions. On peut supposer que ce résultat est engendré par une erreur faite précédemment, lorsque l’on a programmé l’encodeur et le décodeur, ou bien dans les fonctions qui permettent d’évaluer les circuits booléens.

c) Code de Hamming lorsque le bruit est trop important

Pour montrer que le Code de Hamming n’est plus persistant lorsque 2 bits sont corrompus, nous pourrions raisonner comme ci-dessus en implémentant une méthode qui ajoute 2 portes ‘NON’ aux valeurs de sortie de l’encodeur puis effectuer des tests pour vérifier que les codes avant l’encodage et après le décodage sont différents pour toutes les combinaisons possibles.